

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

TURING

图灵程序设计丛书

갈벗

【韩】李在弘 著
武传海 译



DOCKER FOR THE REALLY IMPATIENT

Docker

基础与实战



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

李在弘

目前管理PYRASIS.COM个人网站，编写并发布多种技术文档。曾在NC Software参与开发游戏《天堂永恒》(Lineage Eternal)，并在Ntreev开发移动游戏服务器，还曾负责FFS File System Driver for Windows开源项目。最近正在研究Cocos2d-x移动游戏引擎的Tizen应用，现在主要关注操作系统内核、文件系统、软件开发自动化、游戏引擎、云平台、分布式处理系统。梦想打造个性化的全自动家居，以及设立开源基金会。著有《Windows项目必备实用工具：Subversion、Trac、CruiseControl.NET》《Amazon Web Service技术解析》。

武传海

擅韩语，喜计算机，有多年翻译经验，内容涉及多个领域，尤其擅长翻译各类计算机图书，已出版多部韩语译著。

QQ: 768160125
jeonhae@126.com

TURING

图灵程序设计丛书



Docker

基础与实战

DOCKER FOR THE REALLY IMPATIENT

【韩】李在弘 著
武传海 译

人民邮电出版社
北京

图书在版编目(CIP)数据

Docker基础与实战/(韩)李在弘著;武传海译

-- 北京:人民邮电出版社,2016.6

(图灵程序设计丛书)

ISBN 978-7-115-41962-0

I. ①D… II. ①李… ②武… III. ①Linux操作系统
—程序设计 IV. ①TP316.89

中国版本图书馆CIP数据核字(2016)第049472号

Original Title: 가장 빨리 만나는 도커 (Docker)

Docker for the Really Impatient by Lee, Jae-Hong

Copyright © 2014 Lee, Jae-Hong

Originally published by Gilbut Publishing Co., Ltd.

All rights reserved.

Simplified Chinese copyright © 2016 by POSTS & TELECOM PRESS

This Simplified Chinese edition arranged with Gilbut Publishing Co., Ltd. through Eric Yang Agency

本书中文简体字版由 Gilbut 授权人民邮电出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

内 容 提 要

本书从Docker基础理论出发,更侧重实际业务中的技术与应用。重点在于后半部分在Amazon EC2、Google Colud Platform等平台上的使用方法,以及Rails与Django应用程序构建方法等,都是能够直接运用于实操的技术点。本书是利用Docker构建开发系统、测试系统、操作系统的优秀指南,非常适合一线开发人员。

◆ 著 [韩]李在弘

译 武传海

责任编辑 陈曦

责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

三河市海波印务有限公司印刷

◆ 开本:800×1000 1/16

印张:19.5

字数:423千字

2016年6月第1版

印数:1-3 000册

2016年6月河北第1次印刷

著作权合同登记号 图字:01-2015-1716号

定价:69.00元

读者服务热线:(010)51095186转600 印装质量热线:(010)81055316

反盗版热线:(010)81055315

广告经营许可证:京东工商广字第8052号

与其他领域相比，开源开发环境的变化速度最快。随着 GitHub 的出现，贡献代码的门槛正变得越来越低。我们也可以灵活使用 Git 这一分布式版本管理系统对著名项目进行派生（fork），然后构建自己的项目。

服务器开发及运营环境中，用于隔离目录的 chroot 或在 Linux 内核级别实现容器的 LXC 技术出现已久，但未能得到广泛应用。此时，Docker 利用 GitHub 的共享模型构建了极为便利的平台。

与从 GitHub 派生项目一样，借助 Docker，用户也可以基于 Docker Hub 中的镜像创建并分享自己的镜像。包括主要开源项目的官方镜像在内，其他用户上传的镜像以及自己上传的镜像都可以在全世界范围内共享使用，令人十分惊叹。这种统一接口并将数据集中存放的做法将效率提高到超乎想象的地步。

Linux 与生俱来的局限性是可执行文件与库之间的兼容性问题。针对该问题，多个 Linux 发行版推出了固有的包系统，但仍然无法完美解决。Docker 将应用程序或服务中可运行状态的组合用容器捆绑在一起，再通过网络共享。使用 Docker 可以帮助 Linux 服务器管理员大大缩短在系统构建与管理上浪费的时间，也不会再烦恼编译安装后无法完全删除。

灵活使用 Docker 可以帮助用户脱离从属于特定云平台的环境，只要愿意，就可以从 Amazon Web Service 轻松迁移到 Google Cloud Platform 以及 Microsoft Azure 平台。使用 Docker 镜像无需逐一搭建 Linux 服务器，只要使用原有的 Docker 镜像即可。无论何时都可以轻松转向费用更低、条件更好的服务。

本书并未涵盖 Docker 的所有内容。Docker 目前仍在发展，新的应用方法层出不穷。Docker 并不是一款固定不变的产品，其与多种程序的不同组合会产生无穷无尽的应用方法。因此，与其介绍目前出现的所有应用方法，不如先熟练掌握 Docker 的基本使用方法，不断创建服务并深入探索。

Docker 将服务器开发与运营带入一片全新的天地，下面请随我感受 Docker 的无尽魅力吧！

李在弘

2014 年 11 月

第 1 章 Docker 1

- 1.1 虚拟机与 Docker 3
 - 1.1.1 虚拟机 4
 - 1.1.2 Docker 5
 - 1.1.3 Linux 容器 6
- 1.2 Docker 镜像与容器 8

第 2 章 安装 Docker 11

- 2.1 Linux 11
 - 2.1.1 自动安装脚本 11
 - 2.1.2 Ubuntu 11
 - 2.1.3 RedHat Enterprise Linux、CentOS 12
 - 2.1.4 使用最新二进制文件 12
- 2.2 Mac OS X 13
- 2.3 Windows 16

第 3 章 使用 Docker 23

- 3.1 使用 search 命令搜索镜像 23
- 3.2 使用 pull 命令下载镜像 25
- 3.3 使用 images 命令列出镜像目录 25
- 3.4 使用 run 命令创建容器 25
- 3.5 使用 ps 命令查看容器列表 26
- 3.6 使用 start 命令启动容器 26
- 3.7 使用 restart 命令重启容器 27
- 3.8 使用 attach 命令连接容器 27
- 3.9 使用 exec 命令从外部运行容器内的命令 27
- 3.10 使用 stop 命令终止容器 28
- 3.11 使用 rm 命令删除容器 28
- 3.12 使用 rmi 命令删除镜像 29

第 4 章 创建 Docker 镜像 31

- 4.1 熟悉 Bash 31

- 4.2 编写 Dockerfile 36
- 4.3 使用 build 命令创建镜像 37

第 5 章 查看 Docker 39

- 5.1 使用 history 命令查看镜像历史 39
- 5.2 使用 cp 命令复制文件 40
- 5.3 使用 commit 命令从容器的修改中创建镜像 40
- 5.4 使用 diff 命令检查容器文件的修改 40
- 5.5 使用 inspect 命令查看详细信息 41

第 6 章 灵活使用 Docker 43

- 6.1 搭建 Docker 私有仓库 43
 - 6.1.1 存储镜像数据到本地 43
 - 6.1.2 使用 push 命令上传镜像 44
 - 6.1.3 存储镜像数据到 Amazon S3 45
 - 6.1.4 使用默认认证 46
- 6.2 连接 Docker 的容器 52
- 6.3 连接到其他服务器的 Docker 容器 53
- 6.4 使用 Docker 数据卷 56
- 6.5 使用 Docker 数据卷容器 59
- 6.6 创建 Docker 基础镜像 60
 - 6.6.1 创建 Ubuntu 基础镜像 60
 - 6.6.2 创建 CentOS 基础镜像 61
 - 6.6.3 创建空基础镜像 62
- 6.7 在 Docker 内运行 Docker 64

第 7 章 详细了解 Dockerfile 67

- 7.1 .dockerignore 68
- 7.2 FROM 68
- 7.3 MAINTAINER 69
- 7.4 RUN 69
- 7.5 CMD 70
- 7.6 ENTRYPOINT 71
- 7.7 EXPOSE 73
- 7.8 ENV 73
- 7.9 ADD 74
- 7.10 COPY 76
- 7.11 VOLUME 77
- 7.12 USER 77

7.13 WORKDIR 78

7.14 ONBUILD 79

第 8 章 使用 Docker 部署应用程序 81

8.1 向一台服务器部署应用程序 81

- 8.1.1 在开发者 PC 安装 Git 并创建仓库 82
- 8.1.2 在开发者 PC 中使用 Node.js 编写 Web 服务器 83
- 8.1.3 在开发者 PC 中编写 Dockerfile 文件 84
- 8.1.4 在开发者 PC 中生成 SSH 密钥 85
- 8.1.5 在服务器端安装 Git 并创建仓库 86
- 8.1.6 在服务器中安装 Docker 87
- 8.1.7 在服务器中安装 SSH 密钥 88
- 8.1.8 在服务器中安装 Git Hook 89
- 8.1.9 在开发者 PC 中推送源代码 90

8.2 向多台服务器部署应用程序 91

- 8.2.1 在开发者 PC 安装 Git 并创建仓库 92
- 8.2.2 在开发者 PC 中使用 Node.js 编写 Web 服务器 93
- 8.2.3 在开发者 PC 中编写 Dockerfile 文件 94
- 8.2.4 在开发者 PC 中生成 SSH 密钥 95
- 8.2.5 在部署服务器安装 Git 并创建仓库 96
- 8.2.6 在部署服务器中生成 SSH 密钥 97
- 8.2.7 在部署服务器中安装 Docker 98
- 8.2.8 在部署服务器中安装 Docker 注册服务器 99
- 8.2.9 在部署服务器中安装 SSH 密钥 100
- 8.2.10 在部署服务器中安装 Git Hook 101
- 8.2.11 在应用程序服务器中安装 Docker 103
- 8.2.12 在应用程序服务器中安装 SSH 密钥 104
- 8.2.13 在开发者 PC 中推送源代码 105

第 9 章 Docker 监控 107

- 9.1 编写监控服务器 Dockerfile 108
- 9.2 编写应用程序服务器 Dockerfile 111
- 9.3 在 Web 浏览器中查看图表 114

第 10 章 在 Amazon Web Services 中使用 Docker 117

- 10.1 在 Amazon EC2 中使用 Docker 117
- 10.2 在 AWS Elastic Beanstalk 中使用 Docker 119
 - 10.2.1 在 AWS 控制台部署 Docker 应用程序 119
 - 10.2.2 使用 Docker Hub 公开仓库镜像 129

- 10.2.3 使用 Docker Hub 私有仓库的镜像 131
- 10.2.4 使用 Git 部署 Elastic Beanstalk Docker 应用程序 139

第 11 章 在 Google Cloud Platform 中使用 Docker 145

- 11.1 安装 Google Cloud SDK 145
- 11.2 在 Compute Engine 中使用 Docker 147
- 11.3 在 Container Engine 中使用 Docker 148

第 12 章 使用 Docker Hub 151

- 12.1 加入 Docker Hub 151
- 12.2 使用 push 命令上传镜像 153
- 12.3 创建 Docker Hub 私有仓库 155
- 12.4 使用 Docker Hub Automated Build 157

第 13 章 使用 Docker Remote API 167

- 13.1 使用 Docker Remote API Python 库 169
 - 13.1.1 创建并启动容器 169
 - 13.1.2 创建镜像 173
 - 13.1.3 显示容器列表 175
 - 13.1.4 显示镜像列表 176
 - 13.1.5 其他示例与函数 176
- 13.2 使用 Docker Remote API Python 库进行 HTTPS 通信 187
 - 13.2.1 创建证书 187
 - 13.2.2 使用 Python 库 191

第 14 章 使用 CoreOS 193

- 14.1 在 VirtualBox 中安装 CoreOS 196
 - 使用 systemd 运行服务 205
- 14.2 使用 Vagrant 安装 CoreOS 206
- 14.3 使用 etcd 211
 - 14.3.1 创建 etcd 键与目录 211
 - 14.3.2 输出 etcd 键与目录列表 212
 - 14.3.3 设置自动删除 etcd 键与目录 212
 - 14.3.4 监视 etcd 键 213
 - 14.3.5 etcd 其他命令 214
- 14.4 使用 fleet 214
 - 14.4.1 输出 fleet 机器列表 215
 - 14.4.2 使用 fleet 运行 Unit 215
 - 14.4.3 输出 fleet Unit 列表 217

- 14.4.4 查看 fleet Unit 状态 217
- 14.4.5 测试 fleet 的自动恢复功能 218
- 14.4.6 使用 fleet 专用选项 219
- 14.4.7 灵活使用 fleet Unit 文件模板 222
- 14.4.8 灵活使用 fleet sidekick 模型 224
- 14.4.9 fleet 其他命令 227
- 14.5 在云服务中使用 CoreOS 227
 - 14.5.1 在 Amazon EC2 中使用 CoreOS 227
 - 14.5.2 在 Google Compute Engine 中使用 CoreOS 229

第 15 章 使用 Docker 搭建 WordPress 博客 231

- 15.1 编写 WordPress Dockerfile 文件 232
- 15.2 编写 MySQL 数据库 Dockerfile 文件 233
- 15.3 创建 WordPress 与数据库容器 236

第 16 章 使用 Docker 构建 Ruby on Rails 应用 237

- 16.1 安装 Ruby 与 Rails 238
- 16.2 编写 Rails Dockerfile 240
- 16.3 编写 PostgreSQL 数据库 Dockerfile 文件 245
- 16.4 创建 Rails 与数据库容器 247

第 17 章 使用 Docker 构建 Django 应用 249

- 17.1 安装 Django 250
- 17.2 编写 Django Dockerfile 文件 253
- 17.3 编写 Oracle 数据库 Dockerfile 文件 258
- 17.4 创建 Django 与数据库容器 261

第 18 章 Docker 应用案例 263

- 18.1 与负载均衡相关的自动伸缩 263
- 18.2 整合开发、测试、运营 264
- 18.3 轻松迁移服务 265
- 18.4 用于测试 267

第 19 章 Docker 命令与选项列表 269

- 19.1 attach 270
- 19.2 build 271
- 19.3 Commit 273
- 19.4 cp 273

- 19.5 create 274
- 19.6 diff 277
- 19.7 events 277
- 19.8 exec 278
- 19.9 export 280
- 19.10 history 280
- 19.11 images 281
- 19.12 import 281
- 19.13 info 282
- 19.14 inspect 283
- 19.15 kill 284
- 19.16 load 284
- 19.17 login 285
- 19.18 logout 286
- 19.19 logs 286
- 19.20 port 287
- 19.21 pause 287
- 19.22 ps 287
- 19.23 pull 288
- 19.24 push 289
- 19.25 restart 289
- 19.26 rm 289
- 19.27 rmi 290
- 19.28 run 291
- 19.29 save 296
- 19.30 search 297
- 19.31 start 297
- 19.32 stop 298
- 19.33 tag 298
- 19.34 top 299
- 19.35 unpause 299
- 19.36 version 300
- 19.37 wait 300

第 1 章

DOCK ER

Docker

Docker 是 Docker 公司（原 dotCloud）2013 年 3 月推出的开源容器项目，上市至今已有 3 年，在世界范围内拥有超高人气。

进入 2010 年，服务器市场急速向云环境转移。人们开始更多地租用虚拟服务器，只要缴纳一定租金即可，不需要购买实际的物理服务器。尤其在搭建物理服务器时，服务器硬件的购买及安装都需要耗费相当长的时间。但在云环境下，无论是 1 台还是 1000 台，只需单击几次即可轻松创建虚拟服务器。

创建虚拟服务器后，还要在其中安装各种软件，进行各种设置。如果只有一两台服务器，那么能够轻松进行设置；但随着服务器数量的增加，采用人工设置就难了。因此，在云环境中进行安装与部署存在很大困难。

Linux/Unix 环境中，虽然可以借助沿用至今的 shell 脚本进行自动安装与设置，但这种方式存在一定局限性。使用 shell 脚本很难实现集中式管理功能和其他复杂功能。并且，Linux 环境中需要安装很多应用程序，设置也比较复杂。特别是一些微小的设置可能会对操作系统与服务的稳定性产生影响。

此时出现了“不可变基础设施”（Immutable Infrastructure）这一概念，指的是主机 OS 与服务运行环境（服务器程序、源代码、已编译的二进制文件）分离，只设置一次运行环境，之后不发生变更（Immutable）。也就是说，将服务运行环境创建为镜像后，部署至各服务器运行。此时若更新服务，则运行环境本身不会发生变更，只要重新生成镜像并再次部署即可。就像云平台中对服务器“用过即扔”，不可变基础设施中的服务运行环境镜像也是用过一次后就扔掉。

在虚拟机中安装 Linux 后，可以安装各种服务器程序与 DB，运行已开发的应用程序或网站。将搭建好的虚拟机镜像复制到多台服务器中运行，之后即可用一个镜像不断创建服务器。

虚拟机服务器通常单独运行，也可以使用以服务形式提供的 Amazon Web Services、Microsoft Azure、Google Cloud Platform。

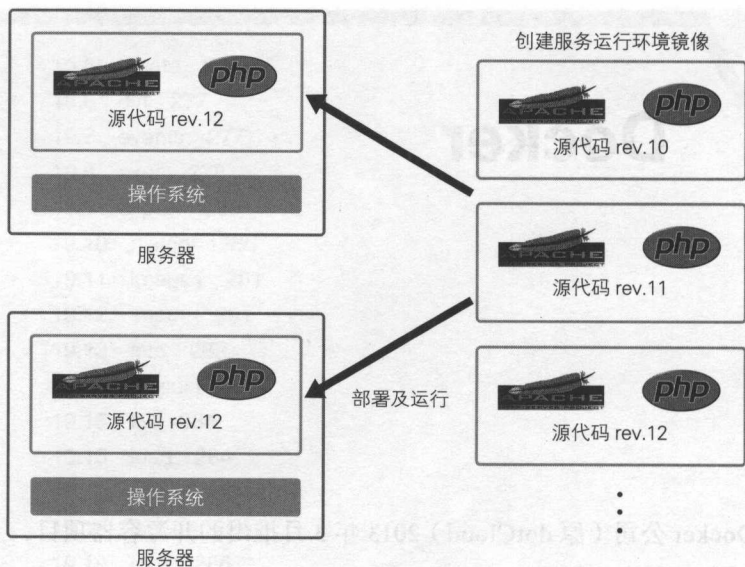


图 1-1 不可变基础设施

不可变基础设施拥有多种优点。

- ▶ 管理方便：由于服务运行环境以镜像形式存在，所以只要管理镜像本身即可。特别是可以集中管理镜像，实现系统部署与管理。此外，镜像生成设置也以文件形式存在，可以灵活用于版本管理系统。
- ▶ 扩展：可以利用一个镜像不断创建服务器。与云平台的自动伸缩功能（Auto Scaling）配合使用，能够轻松实现服务扩展。
- ▶ 测试：只要在开发人员 PC 或测试服务器中运行镜像，就可以搭建与实际服务运行环境一致的环境，非常容易测试。
- ▶ 轻量：分离操作系统与服务运行环境，实现轻量化，提供可以随时运行的环境。

Docker 项目实现了不可变基础设施，本书将详细讲解 Docker 有关内容。

从 Docker 图标与名称本身可以大致猜到 Docker 的功能——一头鲸鱼驮着多个集装箱——这很容易让人联想到服务器运行多个容器（镜像）的场景。

另外，这也意味着 Docker 不仅用于创建并运行镜像，还可以存储与部署（搬运）镜像。“Docker”一词的字典含义为港口（码头）上卸载集装箱的工人，与操纵容器的 Docker 功能类似。

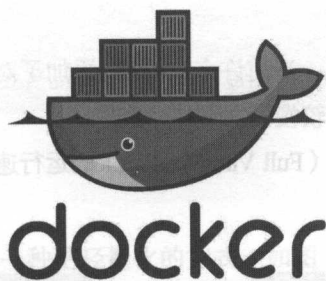


图 1-2 Docker 图标

如同用集装箱装载货物一样，将运行服务所需的所有“元素”全部集中到 Docker 容器之中。这些“元素”可以是常用开源软件，也可以是自己编写的程序。

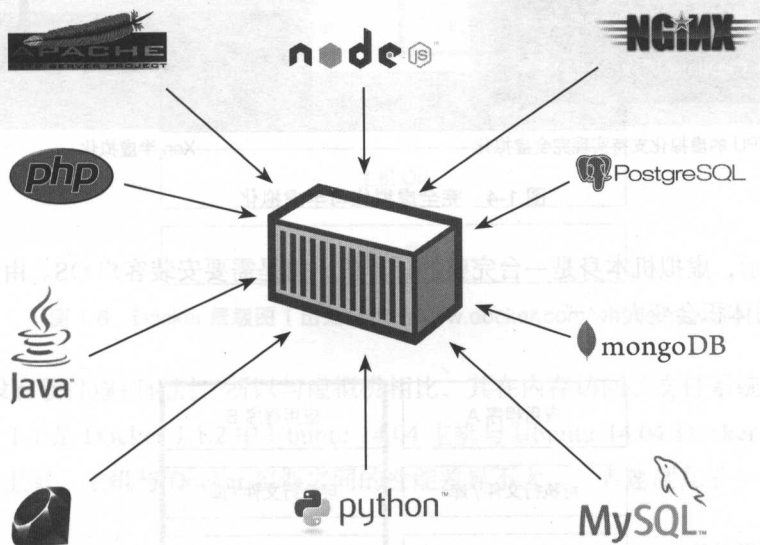


图 1-3 Docker 容器

1.1 ▶ 虚拟机与 Docker

Docker 与我们之前使用的 VMware、Microsoft Hyper-V (Virtual PC)、Xen、Linux KVM 等虚拟机类似。

在虚拟机中安装 Linux 后，可以安装各种服务器程序与 DB，运行已开发的应用程序与网站。将搭建好的虚拟机镜像复制到多台服务器中运行，之后即可用一个镜像不断创建服务器。

虚拟机服务器通常单独运行，也可以使用以服务形式提供的 Amazon Web Services、Microsoft Azure、Google Cloud Platform。

1.1.1 虚拟机

虚拟机非常方便，但性能不佳。当前许多 CPU 都添加了对虚拟化功能的大量支持，但与物理机器相比，虚拟机的运行速度比较慢。

为了进一步改善“完全虚拟化”(Full Virtualization)的运行速度，半虚拟化(Paravirtualization)技术登场，现在正得到广泛应用。

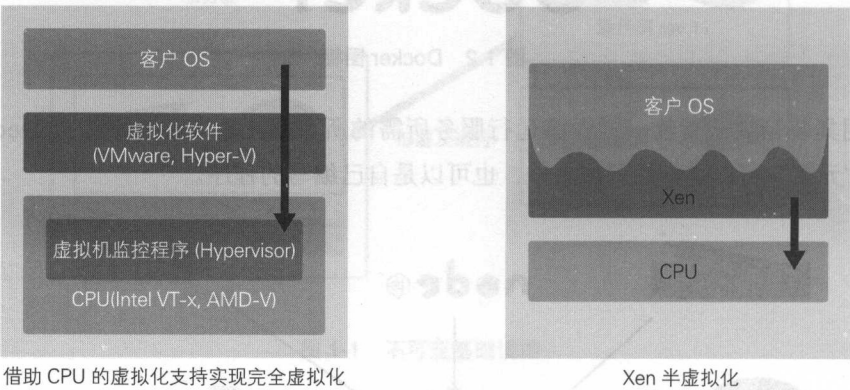


图 1-4 完全虚拟化与半虚拟化

如图 1-5 所示，虚拟机本身是一台完整的计算机，总是需要安装客户 OS。由于镜像中含有 OS，所以镜像的体积会变大。

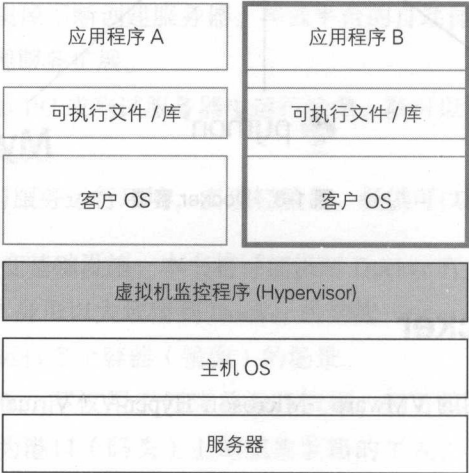


图 1-5 虚拟机层级图 (出处: <http://www.docker.com/whatisdocker/>)

无论网速多快，收发虚拟化镜像都会非常耗时。尤其是开源虚拟化软件，其重点在于 OS 虚拟化，只提供镜像创建与运行功能，在部署与管理功能上存在不足。

提示 虚拟机分部署

对虚拟机进行集中管理与部署的商业产品有 VMware vCenter、Microsoft System Center。

1.1.2 Docker

与半虚拟化相比，Docker 是一种更轻量化的方式。如图 1-6 所示，使用 Docker 则不需要安装客户 OS。Docker 镜像中只隔离并安装服务器运行所需的程序与库，与主机共享 OS 资源（系统调用），这样就大大减小了镜像的体积。

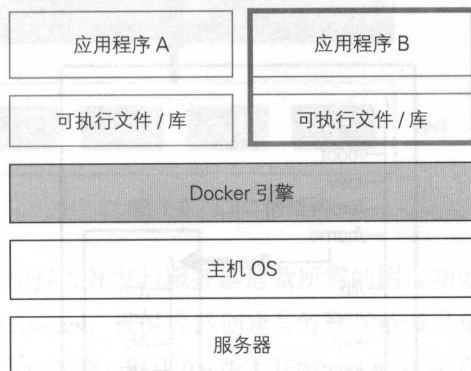


图 1-6 Docker 层级图（出处：<http://www.docker.com/whatisdocker/>）

Docker 没有硬件虚拟化层，所以与虚拟机相比，其在内存访问、文件系统、网络速度上明显快得多。表 1-1 是 Docker 1.1.2 中 Ubuntu 14.04 主机与 Ubuntu 14.04 Docker 容器的性能测试结果。从数值上看，主机与 Docker 容器之间的性能差异不大，二者速度几乎一样。

表 1-1 Docker 1.1.2 中 Ubuntu 14.04 主机与 Ubuntu 14.04 Docker 容器性能测试

	性能测试工具	主机	Docker
CPU	sysbench	1	0.9945
写内存	sysbench	1	0.9826
读内存	sysbench	1	1.0025
磁盘 I/O	dd	1	0.9811
网络	iperf	1	0.9626

与虚拟机不同，Docker 提供了专门创建并部署镜像的功能。如同在 Git 中管理源代码一样，Docker 也提供了镜像版本管理功能。此外，为了进行集中管理，Docker 也提供镜像上传与下载

功能（Push/Pull）。就像 GitHub 一样，Docker Hub 提供帮助用户共享的 Docker 镜像（类似于 Github，也提供个人付费存储服务）。

Docker 提供了多种 API，用户可以轻松实现自动化，这对开发与服务器运营非常有用。

1.1.3 Linux 容器

Linux/Unix 环境一直提供 chroot 命令，用于更改文件系统的根目录（/）。使用 chroot 命令将指定目录设置为根目录后，即可创建 chroot jail（监牢）环境，该环境中不能访问外部文件与目录。由于 chroot 可以这样隔离目录路径，所以使用它可以最大限度地防止服务器信息泄露或受损。

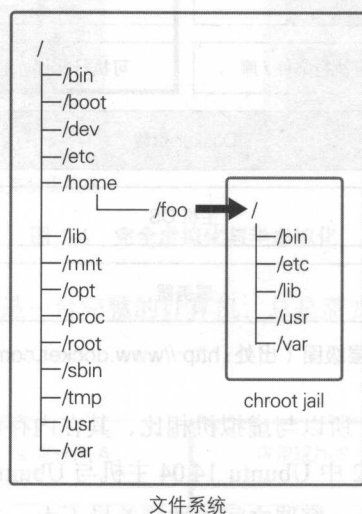


图 1-7 chroot 的目录结构

使用 chroot 命令时，我们必须自己准备要放入 chroot jail 的可执行文件与共享库，设置方法比较复杂。此外，由于不是完美的虚拟环境，故存在诸多制约。后来，Linux 提供了名为 LXC（Linux Container）的系统级虚拟化。

LXC 并不是将整台电脑虚拟化以运行 OS，它是 Linux 内核级别提供了一种隔离虚拟空间。该虚拟空间未安装 OS，所以不能称之为虚拟机，而称作“容器”。

Linux 内核的 Control Groups（cgroups）分配 CPU、内存、磁盘、网络资源，提供完全虚拟化空间。此外，还要隔离进程树、用户账户、文件系统、IPC 等，创建与主机不同的空间，这称为 Namespace isolation(namespaces)。

LXC 利用 Linux 内核的 cgroups 与 namespaces 功能提供虚拟空间。

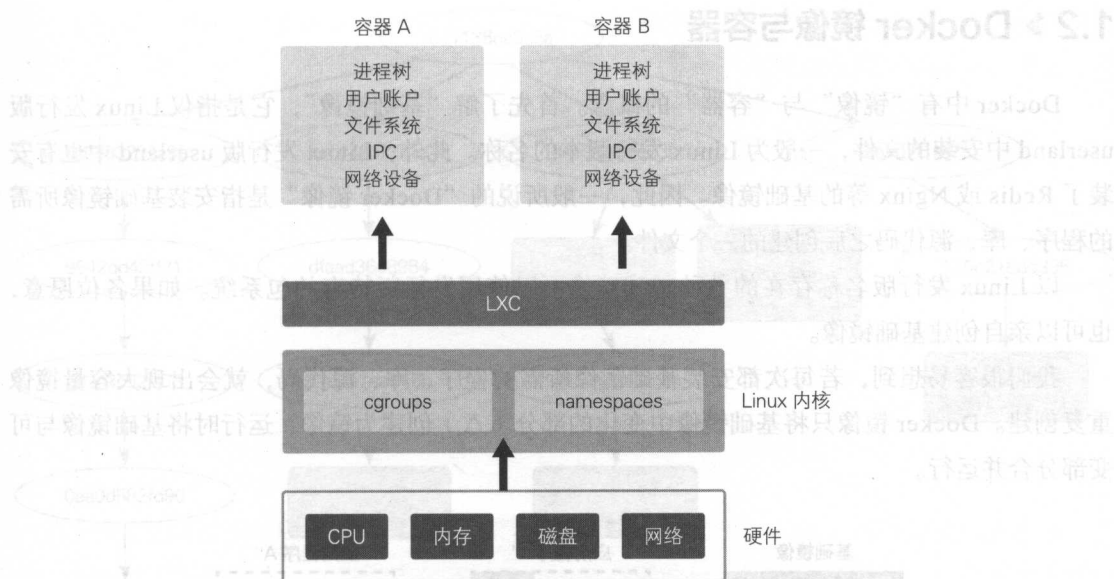


图 1-8 LXC 的结构

LXC 只提供隔离空间，而有关开发与服务器运营所需的附加功能则略有不足。Docker 基于 Linux 内核的 cgroups 与 namespaces，提供容器创建与管理功能及其他附加功能。

开发之初，Docker 基于 LXC 实现；但从 0.9 版本开始，人们开发了 LXC 的替代品 libcontainer。内部称为“运行时驱动”（exec driver），libcontainer 表示为 native，LXC 表示为 lxc。依据运行参数，可以禁用 libcontainer 而使用 LXC。

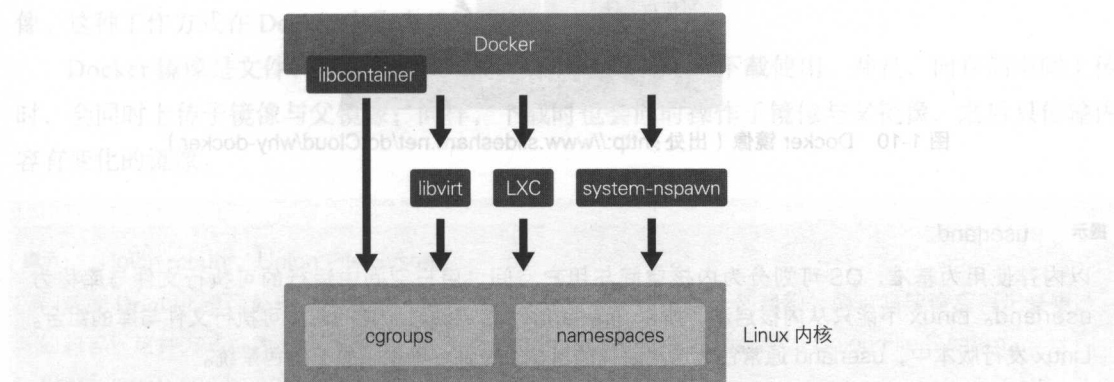


图 1-9 Docker 与 libcontainer

（出处：<http://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer>）

1.2 Docker 镜像与容器

Docker 中有“镜像”与“容器”的概念。首先了解“基础镜像”，它是指仅 Linux 发行版 userland 中安装的文件，一般为 Linux 发行版本的名称。此外，Linux 发行版 userland 中也有安装了 Redis 或 Nginx 等的基础镜像。因此，一般所说的“Docker 镜像”是指安装基础镜像所需的程序、库，源代码之后创建的一个文件。

以 Linux 发行版名称存在的各种基础镜像可以使用发行版特有的包系统。如果各位愿意，也可以亲自创建基础镜像。

我们很容易想到，若每次都安装基础镜像所需的程序、库、源代码，就会出现大容量镜像重复创建。Docker 镜像只将基础镜像中变化的部分（ Δ ）创建为镜像，运行时将基础镜像与可变部分合并运行。

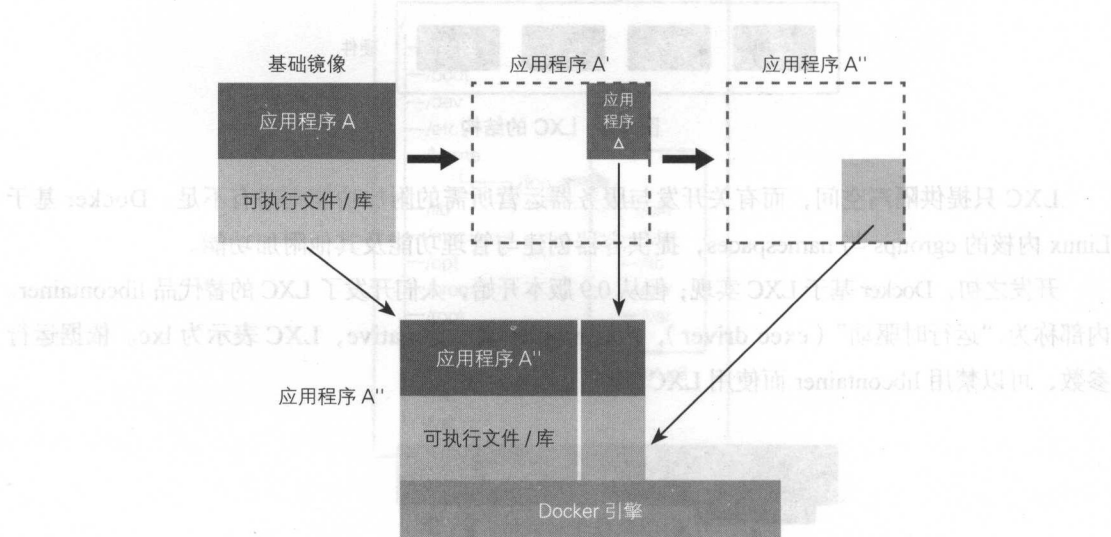


图 1-10 Docker 镜像 (出处: <http://www.slideshare.net/dotCloud/why-docker>)

提示 userland

以内存使用为基准，OS 可划分为内核空间与用户空间，用户空间中运行的可执行文件与库称为 userland。Linux 不能只从内核启动，所以 userland 也指启动时所需的最少可执行文件与库的组合。Linux 发行版本中，userland 通常包含启动所需的可执行文件、库以及原有的包系统。

图 1-11 表示 Docker 镜像之间的依赖关系。Docker 镜像使用 16 进制的 ID 进行区分，各镜像相互独立。centos:centos7 镜像由 511136ea3c5a、34e94e67e63a、b517b77b1a65 组成。向 centos:centos6 镜像设置服务运行所需的程序并创建 Docker 镜像，即得到 example:0.1 镜像。

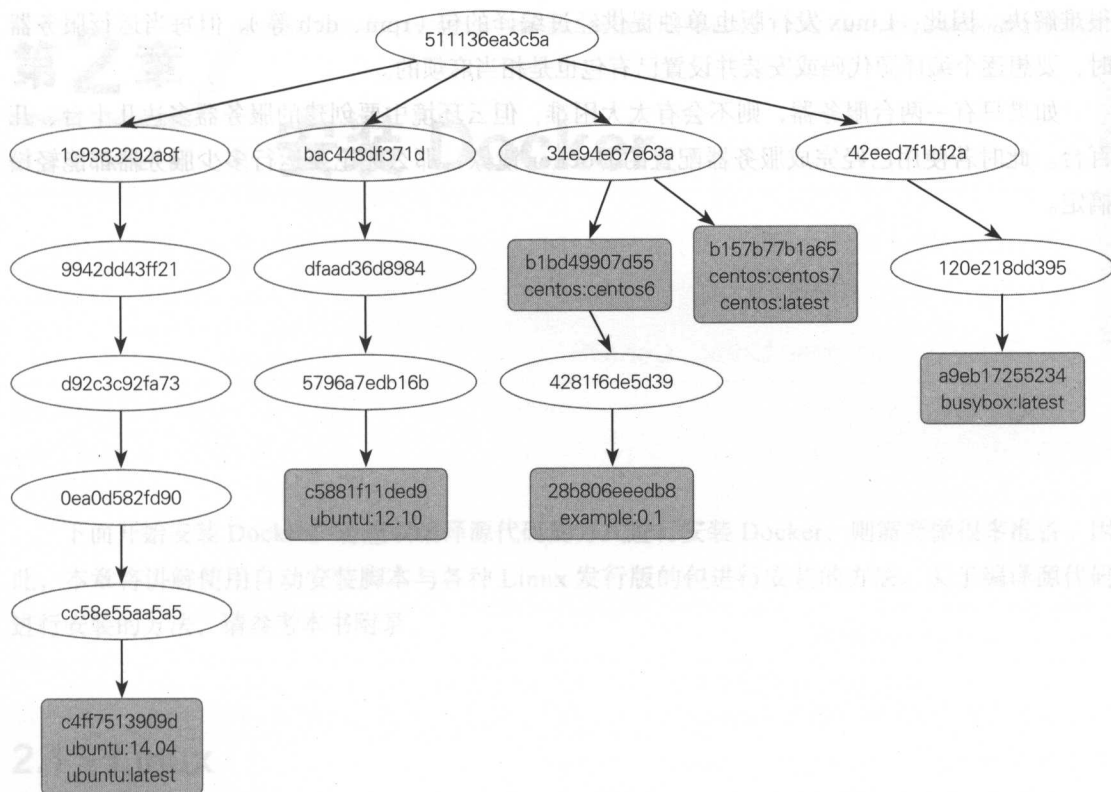


图 1-11 Docker 镜像的依赖关系

也就是说，Docker 不会创建整个镜像，而只针对变化的部分进行创建，然后继续引用父镜像。这种工作方式在 Docker 中称为“层”。

Docker 镜像是文件，所以上传到存储空间后可以在别处下载使用。并且，向存储空间上传时，会同时上传子镜像与父镜像；同样，下载时也会同时操作子镜像与父镜像，之后只传输内容有变化的镜像。

提示 Union mount、Union File System

创建的 Docker 镜像处于只读状态。内容发生变化时不修改镜像，创建额外的可写镜像后再记录更改的内容。这种方式称为 Union mount，支持 Union mount 的文件系统称为 Union File System。

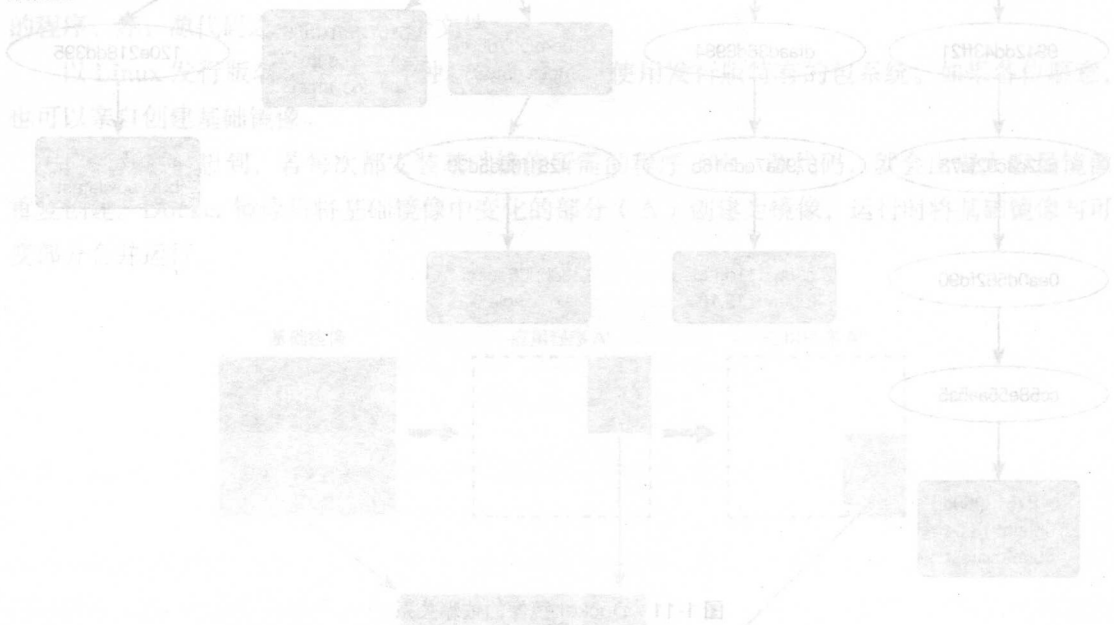
> http://en.wikipedia.org/wiki/Union_mount

Docker 容器是处于运行状态的镜像，使用一个镜像可以创建多个容器。从操作系统角度看，镜像是可执行文件，容器是进程。而已经运行的容器中，也可以将更改的部分创建为镜像。

可以将 Docker 视为特定执行文件或脚本的运行环境。Linux/Unix 系列系统中，文件运行所需的所有组成元素被切分得很小。这样虽然可以使系统结构简单明了，但会导致过度依赖，也

很难解决。因此，Linux 发行版也单独提供经过编译的包（rpm、deb 等）。但每当运行服务器时，要想逐个编译源代码或安装并设置已有包也是相当麻烦的。

如果只有一两台服务器，则不会有太大困难，但云环境中要创建的服务器多达几十台、几百台。此时若使用已经完成服务器配置的 Docker 镜像，那么无论要运行多少服务器都能轻松搞定。



也是说，Docker 不会创建整个镜像，而是只创建变化的部分并挂载到父镜像上。这种工作方式在 Docker 中称为“层”（Layer）。Docker 镜像是文件，所以它们可以像普通文件一样被使用，并且，向存储层上传时，会同时上传千分之一镜像；同样，下载时也会同时下载千分之一镜像。之后只传输内容发生变化的镜像。

Union mount, Union File System

Union mount 是一种文件系统，它允许你将多个文件系统挂载到一个点上，并作为一个单一的文件系统来使用。这在 Docker 中非常有用，因为它允许你将多个镜像挂载到一个容器中，并作为一个单一的文件系统来使用。

在 Docker 中，每个容器都是一个独立的文件系统，它有自己的根目录。这个根目录是由一个或多个镜像组成的。每个容器都有自己的文件系统，但它们都共享同一个底层文件系统。这就是 Union File System 的作用。它允许你将多个文件系统挂载到一个点上，并作为一个单一的文件系统来使用。这在 Docker 中非常有用，因为它允许你将多个镜像挂载到一个容器中，并作为一个单一的文件系统来使用。

第2章

DOCKER

安装 Docker

下面开始安装 Docker。若想以编译源代码的方式进行安装 Docker，则需要做很多准备。因此，本章将讲解使用自动安装脚本与各种 Linux 发行版的包进行安装的方法。关于编译源代码进行安装的方法，请参考本书附录。

2.1 ▶ Linux

Linux 中有两种 Docker 安装方法，一种是使用 Docker 提供的自动安装脚本，另一种是使用 Linux 发行版的 packaging system 直接安装。

2.1.1 自动安装脚本

Docker 自动识别 Linux 发行版的类型，提供用于安装 Docker 包的脚本。

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

使用 get.docker.com 脚本安装 Docker 时，hello-world 镜像也会自动安装。由于不使用 hello-world 镜像，故将其全部删除。

```
$ sudo docker rm $(sudo docker ps -aq)
$ sudo docker rmi hello-world
```

2.1.2 Ubuntu

下列代码表示不使用自动安装脚本而直接在 Ubuntu 中以包进行安装。示例使用的 Ubuntu

版本为 14.01 LTS 64 位。

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

将 /usr/bin/docker.io 可执行文件链接到 /usr/local/bin/docker 并使用。

2.1.3 RedHat Enterprise Linux、CentOS

下面不使用自动安装脚本，而在 RedHat Enterprise Linux (RHEL) 与 CentOS 中直接以包进行安装。由于 RHEL 与 CentOS 6 包仓库中不存在 docker-io，所以使用 EPEL (Fedora Extra Packages For Enterprise Linux) 仓库。

> CentOS 6

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ sudo yum install docker-io
```

由于 AWS EC2 中安装的 Amazon Linux (基于 RHEL) 可以直接使用 EPEL 仓库，所以可以不必安装 epel-release-6-8.noarch.rpm。

在 CentOS 7 中直接安装 docker 包即可。

> CentOS 7

```
$ sudo yum install docker
```

> 启动 Docker 服务

```
$ sudo service docker start
```

> 启动时自动运行

```
$ sudo chkconfig docker on
```

2.1.4 使用最新二进制文件

对于发行版本陈旧或 CentOS 等版本更新慢的发行版，Docker 的包版本通常都较低。下面介绍不使用发行包时，如何直接使用编译后的二进制文件进行安装。

已经采用包方式安装时，可以使用如下命令运行。

```
$ sudo service docker stop
$ sudo wget https://get.docker.com/builds/Linux/x86_64/docker-latest \
  -O $(type -P docker)
$ sudo service docker start
```

使用如下命令进行全新安装。

```
$ wget https://get.docker.com/builds/Linux/x86_64/docker-latest
$ chmod +x docker-latest
$ sudo mv docker-latest /usr/local/bin/docker
$ sudo /usr/local/bin/docker -d
```

在 URL 中指定 `docker-latest` 后，会下载最新版本。若对版本进行了指定——如 `docker-1.3.0`——则只下载所指定的版本。

2.2 ▶ Mac OS X

在 Mac OS X 中利用 Boot2Docker 使用 Docker。从以下 URL 下载 .pkg 文件。

▶ <https://github.com/boot2docker/osx-installer/releases>

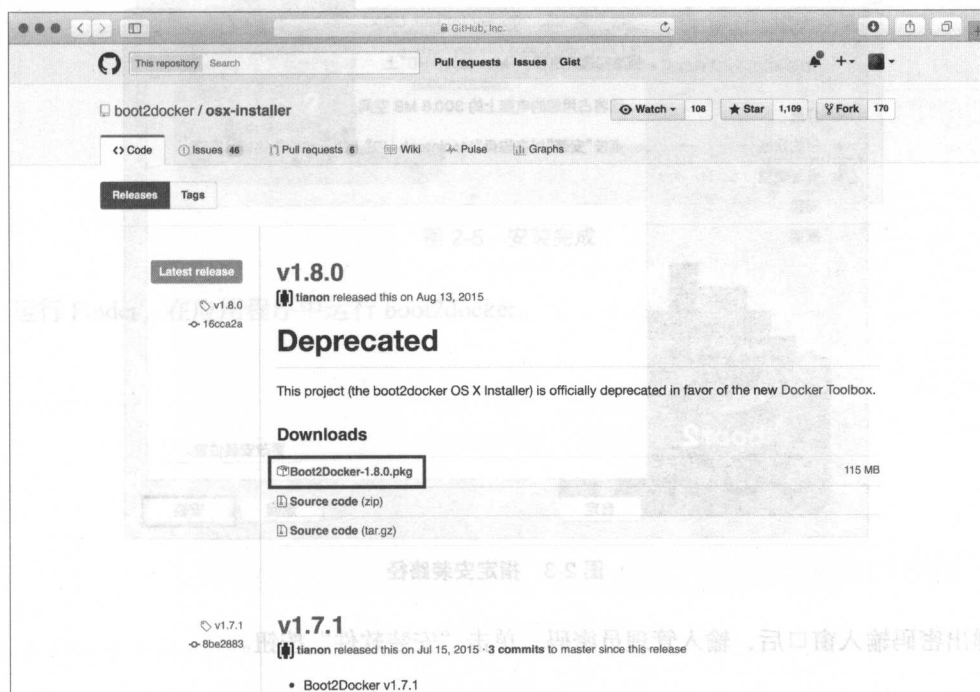


图 2-1 从 GitHub 下载 .pkg 文件

文件下载完成后, 运行 .pkg 文件。出现安装界面, 单击“继续”按钮进行安装。

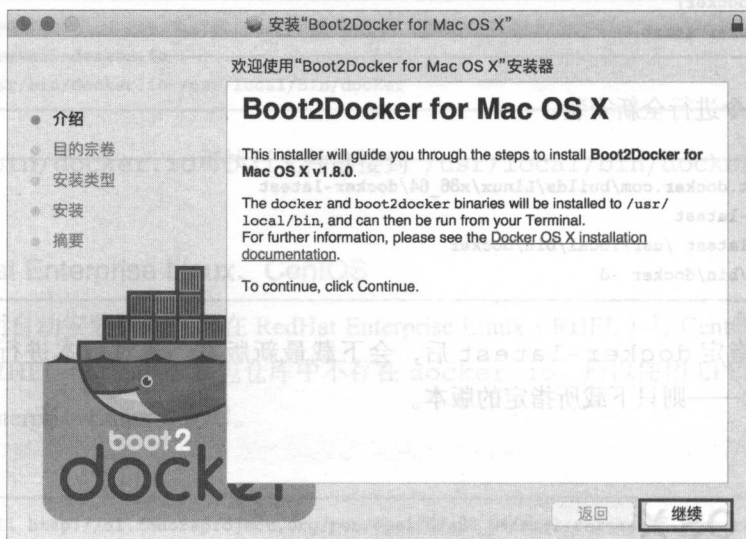


图 2-2 开始安装 Boot2Docker

下面指定 Boot2Docker 的安装路径, 采用默认安装路径。

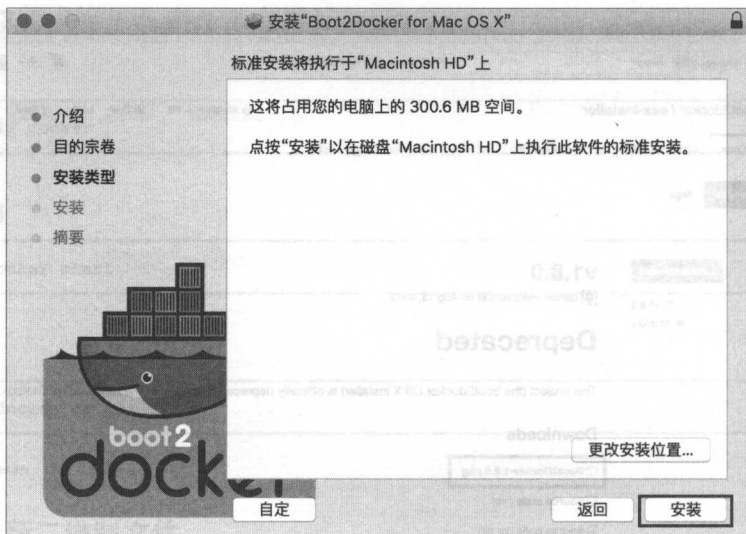


图 2-3 指定安装路径

弹出密码输入窗口后, 输入管理员密码, 单击“安装软件”按钮。

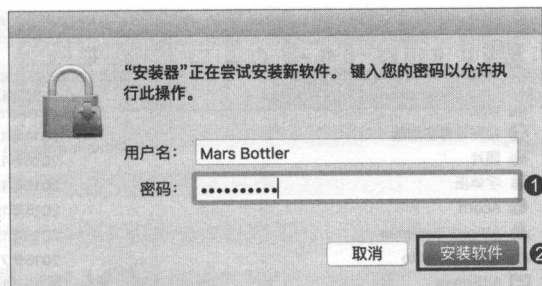


图 2-4 输入管理员密码

安装完成后，单击“关闭”按钮。

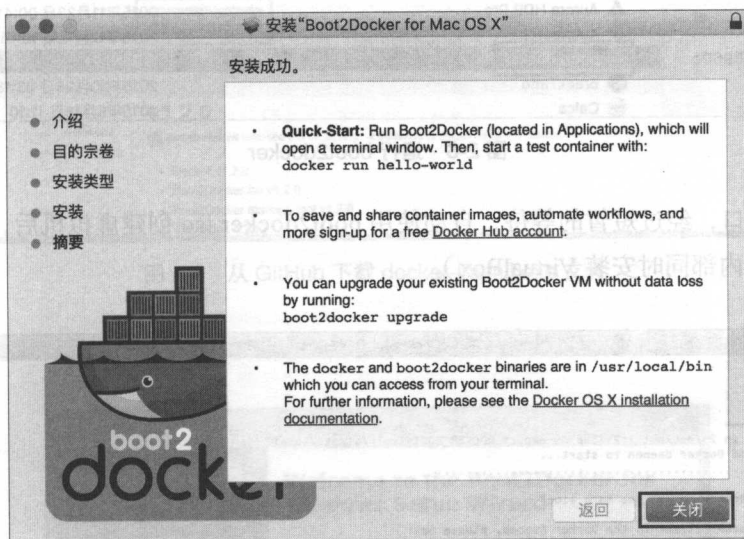


图 2-5 安装完成

运行 Finder，在应用程序中运行 boot2docker。



图 2-6 运行 boot2docker

出现终端窗口，经过短暂的等待，自动使用 boot2docker.iso 创建虚拟机后，连接到虚拟机（Boot2Docker 在内部同时安装 VirtualBox）。

```
pyrasis — Boot2Docker for OSX — bash — 131x25

...+...
...+oE...
...=...

/bin/boot2docker ip 2>/dev/null:2375up && export DOCKER_HOST=tcp://$(/usr/local
Waiting for VM and Docker daemon to start...
.....
Started.
Trying to get Docker socket one more time

To connect the Docker client to the Docker daemon, please set:
export DOCKER_HOST=tcp://192.168.59.103:2375

bash-3.2$ docker version
Client version: 1.3.0
Client API version: 1.15
Go version (client): go1.3.3
Git commit (client): c78088f
OS/Arch (client): darwin/amd64
Server version: 1.3.0
Server API version: 1.15
Go version (server): go1.3.3
Git commit (server): c78088f
bash-3.2$
```

图 2-7 虚拟机创建完成

2.3 Windows

在 Windows 中使用 Boot2Docker 安装 Docker。从以下 URL 下载 docker-install.exe 安装文件。

➤ <https://github.com/boot2docker/windows-installer/releases>

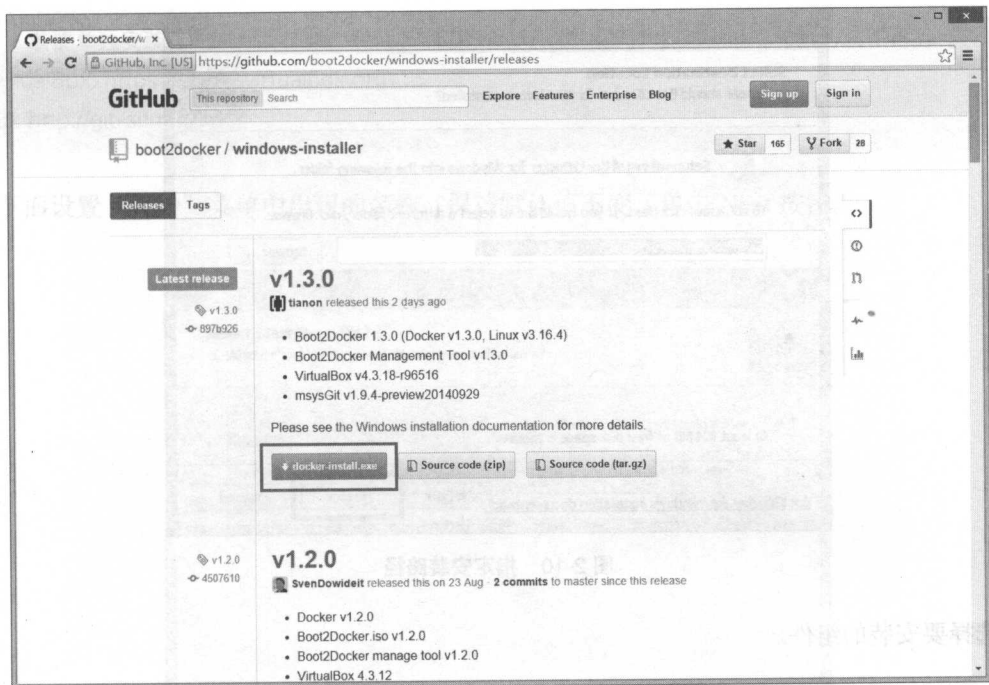


图 2-8 从 GitHub 下载 docker-install.exe 文件

文件下载完成后，运行 docker-install.exe 文件。出现安装画面，单击 Next 按钮。

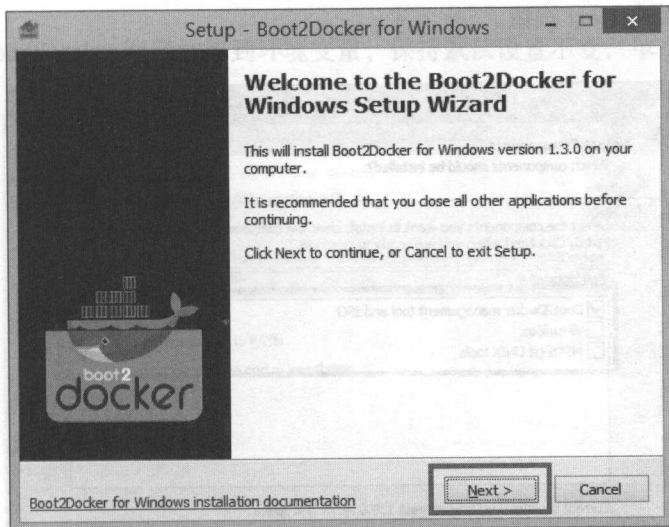


图 2-9 开始安装 Boot2Docker

指定 Boot2Docker 的安装路径，采用默认设置。

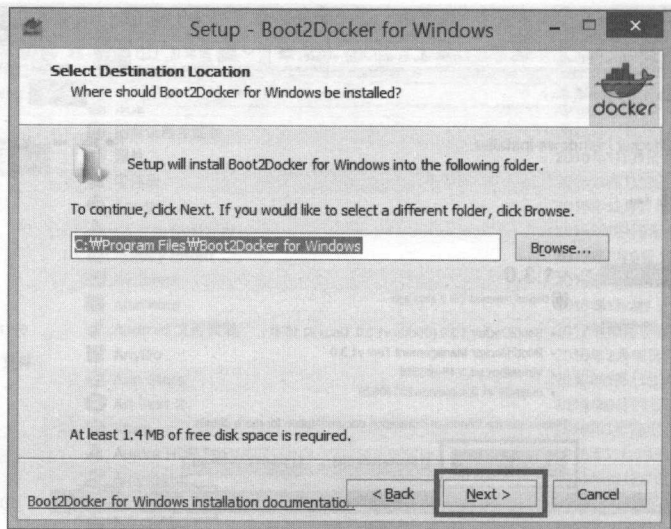


图 2-10 指定安装路径

选择要安装的组件。

- Boot2Docker Management script and ISO : 因为要安装 Boot2Docker, 所以该项必选。
- VirtualBox : 若未安装 VirtualBox, 则选择该项。
- MSYS-git UNIX tools : 若未安装用于 Windows 的 Git, 则选择该项。

选择完成后, 单击 Next 按钮。

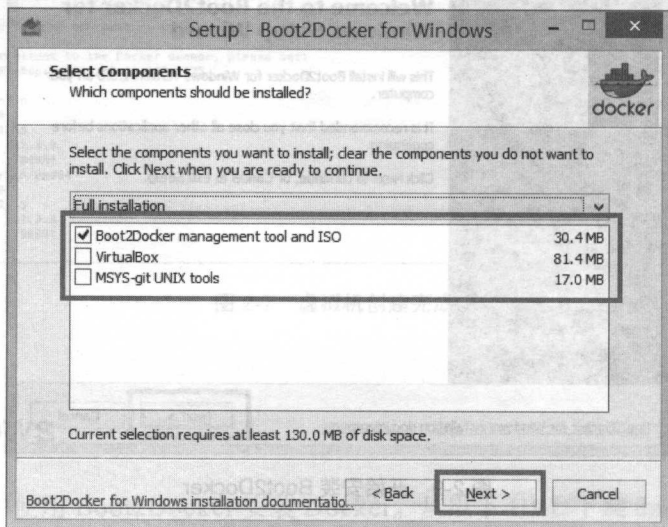


图 2-11 选择安装组件

提示 若想单独安装 VirtualBox 与 Git，可以从以下网址下载安装文件进行安装。安装方法不再另行说明。

- VirtualBox: <https://www.virtualbox.org>
- Git: <http://git-scm.com>

下面设置“开始”菜单中出现的名称，保持默认值不变，单击 Next 按钮。

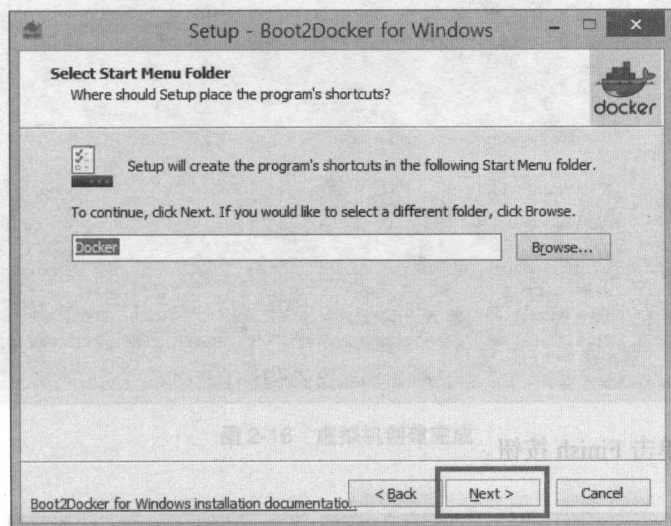


图 2-12 设置“开始”菜单

设置是否将 Boot2Docker 路径注册到环境变量，保持默认设置不变，单击 Next 按钮。

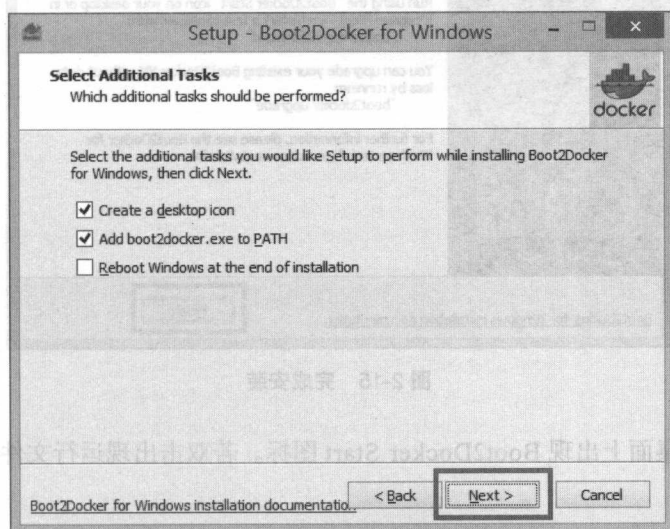


图 2-13 设置环境变量

单击 Install 按钮，安装 Boot2Docker。

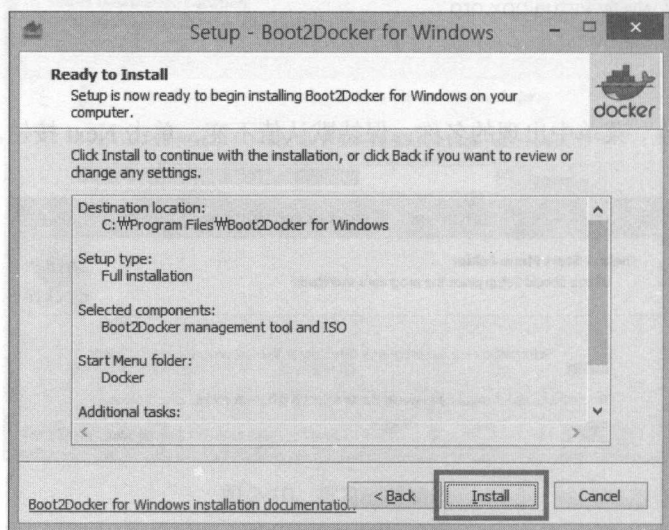


图 2-14 安装程序

安装完成后，单击 Finish 按钮。

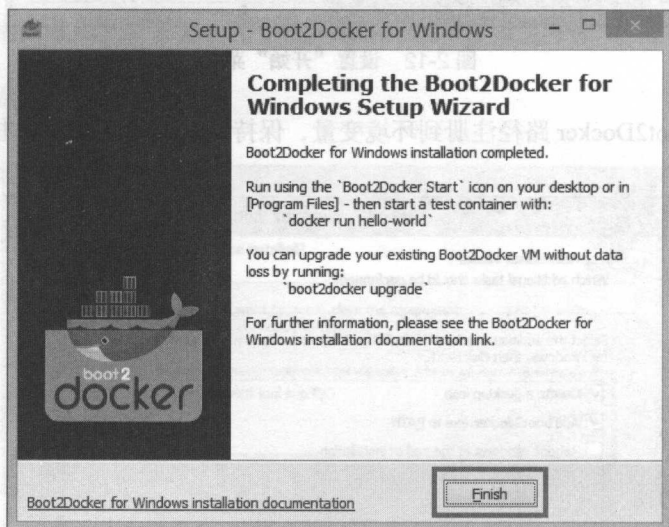


图 2-15 完成安装

安装完毕后，桌面上出现 Boot2Docker Start 图标。若双击出现运行文件选择窗口，则选择 Git Bash 的 sh.exe。

出现命令行窗口，经过短暂的等待，自动使用 boot2docker.iso 创建虚拟机，之后连接到虚

第3章

DOCKER

使用 Docker

Docker 命令格式为 `docker< 命令 >`，比如 `docker run`、`docker push`，且必须总是以 root 权限运行。

学习 Docker 的基本用法之前，先从 Docker Hub 下载并运行镜像。

3.1 ▶ 使用 search 命令搜索镜像

Docker 通过 Docker Hub (<https://registry.hub.docker.com>) 搭建镜像共享生态系统。著名的 Linux 发行版与开源项目 (Redis、Nginx 等) 的 Docker 镜像都可以在 Docker Hub 中找到，与镜像相关的所有命令默认设置都可以使用 Docker Hub。

使用 `docker search` 命令在 Docker Hub 中搜索镜像。

```
$ sudo docker search ubuntu
```

NAME	DESCRIPTION	STARS	OFFICIAL	AVTOMATED
ubuntu	Official Ubuntu base image	383		
stackbrew/ubuntu	Official Ubuntu base image	40		
crashsystems/gitlab-docker	A trusted, regularly updated build of GitL...	19		[OK]
dockerfile/ubuntu	Trusted Ubuntu (http://www.ubuntu.com/) Bu...	15		[OK]
ubuntu-upstart	Upstart is an event-based replacement for ...	7		
cmfatih/phantomjs	PhantomJS [phantomjs 1.9.7, casperjs 1.1...	5		[OK]
dockerfile/ubuntu-desktop	Trusted Ubuntu Desktop (LXDE) (http://lxde...	5		[OK]
lukasz/docker-scala	Dockerfile for installing Scala 2.10.3, Ja...	5		[OK]
litaio/ruby	Ubuntu 14.04 with Ruby 2.1.2 compiled from...	5		[OK]

输入命令后查找到多个镜像。一般带有 `ubuntu`、`centos`、`redis` 等 OS 或程序名称的镜像都是官方镜像，其他都是用户自己创建并共享的镜像。

在 Docker Hub 中搜寻镜像后, 查看相关镜像的 Tags 选项卡, 可以看到当前可用的镜像版本。

➤ https://registry.hub.docker.com/_/ubuntu/tags/manage/

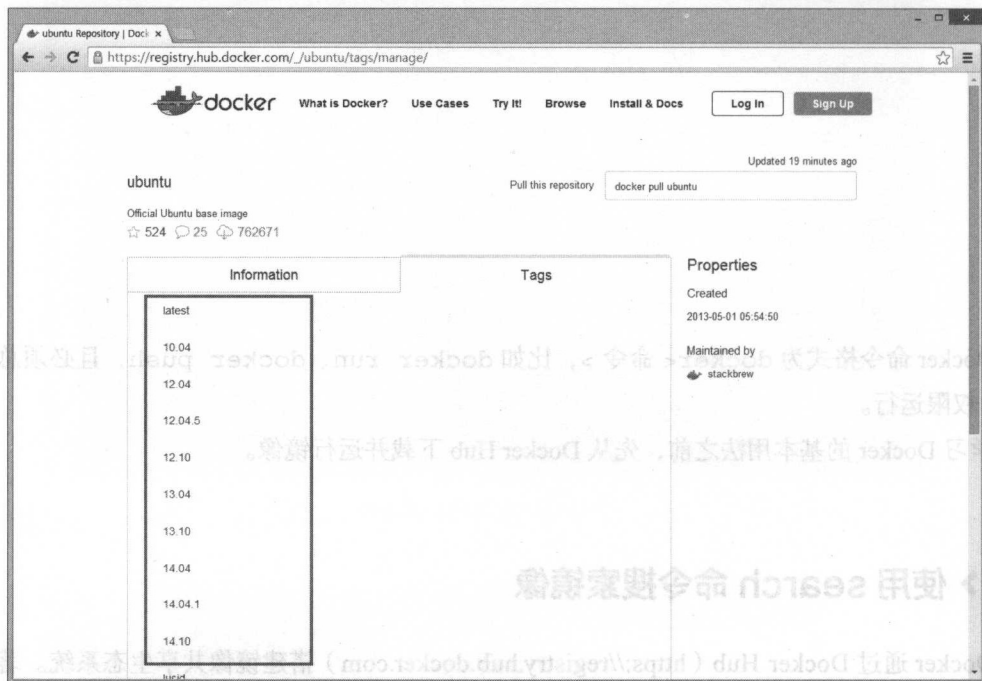


图 3-1 Ubuntu Linux 镜像的标签列表

提示 不输入 sudo

执行 docker 命令时必须使用 root 权限, 所以普通用户总是要输入 sudo。每次都输入 sudo 很麻烦, 还经常会忘记。有两种方法可以不用输入 sudo。

- 一开始就以 root 账户登录或者使用 `sudo su` 命令切换至 root 账户

```
$ sudo su
#
```

- 将当前账户包含到 docker 组。(docker 组与 root 权限是一样的, 请只包含必需的账户。)

```
$ sudo usermod -aG docker ${USER}
$ sudo service docker restart
```

退出当前账户, 重新登录。

3.2 ▶ 使用 pull 命令下载镜像

下面从 Docker Hub 下载 Ubuntu Linux 镜像。

```
$ sudo docker pull ubuntu:latest
```

命令格式为 `docker pull < 镜像名称 >:< 标签 >`。设置 `latest` 后下载最新版本，也可以直接指定具体的版本号，比如 `ubuntu:14.04`、`ubuntu:12.10`。

镜像名称中，在“/”之前指定用户名——比如 `pyrasis/ubuntu`——则可以从 Docker Hub 下载指定用户上传的镜像。官方镜像名称中不会出现用户名。

提示 主机中安装的 Linux 发行版可以与 Docker 镜像的版本不同。CentOS 中可以运行 Ubuntu 容器。

3.3 ▶ 使用 images 命令列出镜像目录

下面列出本地主机中已经下载的镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	latest	e54ca5efa2e9	Less than a second ago	276.1 MB

`docker images` 命令用于列出本地主机中所有镜像。在该命令中设置镜像名称——比如 `docker images ubuntu`——则只列出名称相同但标签不同的镜像。

3.4 ▶ 使用 run 命令创建容器

使用镜像创建容器后，运行 Bash shell。

```
$ sudo docker run -i -t --name hello ubuntu /bin/bash
```

命令格式为 `docker run< 选项 >< 镜像名称 >< 要运行的文件 >`。将 `ubuntu` 镜像创建为容器后，运行 `ubuntu` 镜像中的 `/bin/bash`。也可以不用镜像名称而用镜像 ID。

- ▶ 使用 `-i(interactive)`、`-t(Pseudo-tty)` 选项可以在运行的 Bash shell 中进行输入与输出。
- ▶ 使用 `--name` 选项可以指定容器名称。若不指定名称，Docker 会自动生成名称并进行指定。

现已生成与主机 OS 完全隔离的空间。使用 `cd`、`ls` 命令查看容器内部，可以发现，其与主机 OS 不同。输入 `exit` 命令从 Bash shell 退出。因为在 Ubuntu 镜像中直接运行 `/bin/bash` 可执行文件，所以退出后容器也会终止（stop）。

提示 若 CentOS 中出现如下错误

```
unable to remount sys readonly: unable to mount sys as readonly max retries reached
```

向 `/etc/sysconfig/docker` 文件如下添加 `--exec-driver=libc`。

```
> /etc/sysconfig/docker
```

```
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon process
# These will be parsed by the sysv initscript and appended
# to the arguments list passed to docker -d
```

```
other_args="--selinux-enabled --exec-driver=libc"
```

重启 Docker 服务。

```
$ sudo service docker restart
```

3.5 ▶ 使用 ps 命令查看容器列表

输入如下命令，列出所有容器。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6338ce52d07c	ubuntu:latest	/bin/bash	4 seconds ago	Exited (0) Less than a second ago		hello

命令格式为 `docker ps`。使用 `-a` 选项同时列出终止的容器，否则只列出当前运行的容器。由于前面创建容器时设置了名称为 `hello`，所以容器列表中出现 `hello` 字样。

3.6 ▶ 使用 start 命令启动容器

使用如下命令可以重启刚刚终止的容器。

```
$ sudo docker start hello
```

命令格式为 `docker start< 容器名称 >`，也可以使用容器 ID 代替容器名称。

输入如下命令，列出正在运行的容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6338ce52d07c	ubuntu:latest	/bin/bash	15 minutes ago	Up 3 seconds		hello

启动 hello 容器。

3.7 > 使用 restart 命令重启容器

与重启 OS 类似，也可以使用如下命令重启某个容器。

```
$ sudo docker restart hello
```

命令格式为 `docker restart< 容器名称 >`，也可以使用容器 ID 代替容器名称。

3.8 > 使用 attach 命令连接容器

下面连接一个正在运行的容器。执行如下命令后，再输入一次回车键，显示 Bash shell。

```
$ sudo docker attach hello
root@6338ce52d07c:/#
```

命令格式为 `docker attach< 容器名称 >`，也可以使用容器 ID 代替容器名称。

虽然运行 `/bin/bash` 后可以自由输入命令，但如果运行 DB 或服务器应用程序将无法进行输入，只能看到输出。

在 Bash shell 中输入 `exit` 或 `Ctrl+D` 终止容器。若依次输入 `Ctrl+P`、`Ctrl+Q`，则不会终止容器而只退出。

3.9 > 使用 exec 命令从外部运行容器内的命令

当前容器正以 `/bin/bash` 形式处于运行状态，也可以不通过 `/bin/bash` 而从外部运行容器内的命令。


```
$ sudo docker exec hello echo "Hello World"
Hello World
```

命令格式为 `docker exec < 容器名称 > < 命令 > < 形式参数 >`，也可以使用容器 ID 代替容器名称。该命令只在容器运行时可用，容器处于终止状态时无法使用该命令。

运行容器内的 `echo` 命令，形式参数指定为 `Hello World`，故输出 `Hello World`。在已经运行的容器中使用 `apt-get`、`yum` 命令安装包或运行各种守护进程时，常常使用 `docker exec` 命令。

3.10 ▶ 使用 stop 命令终止容器

下面终止容器。首先列出正在运行的容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6338ce52d07c	ubuntu:latest	/bin/bash	51 minutes ago	Up 2 minutes		hello

使用如下命令终止容器。

```
$ sudo docker stop hello
```

命令格式为 `docker stop < 容器名称 >`，也可以使用容器 ID 代替容器名称。

列出正在运行的容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

由于 `hello` 容器终止，故不显示任何内容。

3.11 ▶ 使用 rm 命令删除容器

下面删除已创建的容器。

```
$ sudo docker rm hello
```

命令格式为 `docker rm < 容器名称 >`，也可以使用容器 ID 代替容器名称。

列出所有容器。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

由于删除了 hello 容器，故不显示任何内容。

3.12 ▸ 使用 rmi 命令删除镜像

使用如下命令删除指定镜像。

```
$ sudo docker rmi ubuntu:latest
```

命令格式为 `docker rmi < 镜像名称 > : < 标签 >`，也可以使用容器 ID 代替容器名称。像 `docker rmi ubuntu` 一样，只要指定镜像名称，所有带 `ubuntu` 名称的镜像都会被删除，无论标签是否相同。

显示镜像列表。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
------------	-----	----------	---------	--------------

由于删除了 `ubuntu` 镜像，故不显示任何内容。

第4章

DOCKER

创建 Docker 镜像

我们已经学习了基本镜像与容器的使用方法，下面学习如何创建镜像。

4.1 ▶ 熟悉 Bash

由于 Docker 是基于 Linux 实现的，所以创建镜像时主要使用 Bash（Bourne-again shell）。创建镜像前，先简单了解常用 Bash 语法。

表 4-1 Bash 基本语法

语法	说明
>	输出重定向。将命令执行的标准输出（stdout）保存为文件。Unix 系列操作系统将设备视为文件，故可以将命令执行结果发送到特定设备 <pre>\$ echo "hello" > ./hello.txt \$ echo "hello" > /dev/null</pre>
<	输入重定向。读取文件内容，用作命令的标准输入（stdin） <pre>\$ cat < ./hello.txt</pre>
>>	将命令执行的标准输出（stdout）添加到文件。“>”会覆盖文件原有内容，而“>>”仅将内容添加到文件末尾 <pre>\$ echo "world" >> ./hello.txt</pre>
2>	将命令执行的标准错误（stderr）保存为文件
2>>	将命令执行的标准错误（stderr）添加到文件
&>	将标准输出与标准错误全部保存为文件

(续)

语法	说明
1>&2	<p>将标准输出发送为标准错误。虽然使用 <code>echo</code> 命令可以将字符串输出为标准输出，但由于发送为标准错误，故变量中没有字符串</p> <pre>\$ hello=\$(echo "Hello World" 1>&2) \$ echo \$hello</pre>
2>&1	<p>将标准错误发送为标准输出。由于不存在 <code>abcd</code> 这样的命令，所以发生的错误发送为标准输出后，再次发送至 <code>/dev/null</code>，故不输出任何内容</p> <pre>\$ abcd > /dev/null 2>&1</pre>
	<p>管道。将命令执行的标准输出发送为其他命令的标准输入，即在第二个命令中处理第一个命令的输出值</p> <pre>\$ ls -al grep .txt</pre>
\$	<p>Bash 的变量。存储值时不使用 <code>\$</code> 符号，仅输入变量时才使用</p> <pre>\$ hello= "Hello World" \$ echo \$hello Hello World</pre>
\$()	<p>将命令的执行结果变量化。将命令的执行结果存储到变量或者传递给其他命令的形式参数时使用。此外，也用于将命令的执行结果放入字符串</p> <pre>\$ sudo docker rm \$(docker ps -aq) \$ echo \$(date) Tue Sep 9 21:24:30 KST 2014</pre>
` `	<p>与 <code>\$()</code> 类似，将命令执行结果变量化</p> <pre>\$ sudo docker rm `docker ps -aq` \$ echo `date` Tue Sep 9 21:24:30 KST 2014</pre>
&&	<p>在一行内运行多个命令。只有前面的命令执行没有错误时，才执行后面的命令</p> <pre>\$ make && make install</pre>
;	<p>在一行内运行多个命令。即使前面的命令执行失败，也会执行后面的命令</p> <pre>\$ false; echo "Hello" Hello</pre>
' '	<p>字符串。不处理 <code>'</code> 中的变量，只使用变量名。也不会处理 <code>"</code> 与 <code>\$()</code>，直接使用</p> <pre>\$ echo '\$USER' \$USER 直接输出 \$USER</pre>

(续)

语法	说明
while	<p>while 循环语句</p> <pre>while : do echo "Hello World"; sleep 1; done</pre>
<<<	<p>将字符串发送到命令（进程）的标准输入</p> <pre>\$ cat <<< "User name is \$USER" User name is pyrrasis</pre>
<<EOF EOF	<p>将多行字符串发送到命令（进程）的标准输入</p> <pre>cat > ./hello.txt <<EOF Hello World Host name is \$(hostname) User name is \$(USER) EOF</pre> <p>cat 命令用于输出文件或标准输入的内容。将 cat 的标准输出保存到此文件，用 <<EOF 将字符串发送到 cat 的标准输入。之后，3 行字符串被保存到此文件</p>
export	<p>设置环境变量，格式为 export <变量>=<值></p> <pre>\$ export HELLO=world</pre>
printf	<p>以指定格式输出值。与管道配合，向命令（进程）输入值</p> <pre>\$ printf 80\nexampleuser\n example-config Port: 80 User: exampleuser Save Configuration (y/n): y</pre> <p>比如，example-config 接收用户输入的 Port、User、Save Configuration。使用 printf 提前设置值，通过管道传递给 example-config 后，即使用户不输入，也会自动输入值。换行符为 \n</p>
sed	<p>更改文本文件中的指定字符串。比如从 hello.txt 文件中查找 hello 字符串，并替换为 world 字符串，执行命令如下所示：</p> <pre>\$ sed -i "s/hello/world/g" hello.txt</pre> <p>命令格式为 sed -i "s/<待查找字符串><替换字符串>/g" <文件名>。与 / 这些特殊字符类似，要在前面添加 \ 字符，即输入时为 \</p>
#	<p>注释。向脚本添加说明或使命令不被执行</p> <pre># echo "Hello World"</pre>

4.2 ▸ 编写 Dockerfile

Dockerfile 是 Docker 镜像设置文件。Dockerfile 中设置的内容会用于创建镜像。

创建并转移到 example 目录。

```
~$ mkdir example
~$ cd example
```

将如下内容保存为 Dockerfile 文件。

▸ dockerbook/Chapter04/Dockerfile

▸ example/Dockerfile

```
FROM ubuntu:14.04
MAINTAINER Foo Bar <foo@bar.com>

RUN apt-get update
RUN apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
RUN chown -R www-data:www-data /var/lib/nginx

VOLUME ["/data", "/etc/nginx/site-enabled", "/var/log/nginx"]

WORKDIR /etc/nginx

CMD ["nginx"]

EXPOSE 80
EXPOSE 443
```

上述示例基于 Ubuntu 14.04 创建 Docker 镜像，且安装 nginx 服务器。

▸ FROM：指定基于的基础镜像。Docker 镜像基于已创建的镜像。指令格式为 < 镜像名称 > : < 标签 >。

▸ MAINTAINER：维护者信息。

▸ RUN：运行 shell 脚本或命令。

» 由于创建镜像的过程中不能接收用户输入，所以在 apt-get install 命令中使用 -y 选项（yum install 也一样）。

» 其他内容是 nginx 设置。

» VOLUME：要与主机共享的目录。也可以在 docker run 命令中使用 -v 选项进行设置。例如，-v /root/data:/data 将主机的 /root/data 目录连接到 Docker 容器的 /data 目录。

▸ CMD：指定容器启动时执行的文件或 shell 脚本。

- › WORKDIR: 为 CMD 中设置的可执行文件设置运行目录。
- › EXPOSE: 与主机相连的端口号。

4.3 ▶ 使用 build 命令创建镜像

Dockerfile 文件编写完成后, 在保存 Dockerfile 文件的 example 目录中执行如下命令。

```
~/example$ sudo docker build --tag hello:0.1 .
```

命令格式为 `docker build <选项> <Dockerfile 路径>`。使用 `--tag` 选项可以设置镜像名称与标签。若只设置镜像名称, 标签就会设置为 `latest`。

稍等片刻即可生成镜像文件, 显示镜像目录。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	e54ca5efa2e9	Less than a second ago	276.1 MB
ubuntu	latest	e54ca5efa2e9	Less than a second ago	276.1 MB
hello	0.1	2031ee0736e8	9 minutes ago	298.4 MB

至此, `hello:0.1` 镜像已经创建, 下面尝试运行。

```
$ sudo docker run --name hello-nginx -d -p 80:80 -v /root/data:/data hello:0.1
```

- › `-d` 选项在后台运行容器。
- › `-p 80:80` 选项将主机的 80 号端口与容器的 80 号端口连接起来, 并暴露到外部。这样设置后, 连接 `http://<主机 IP>:80` 就会连接到容器的 80 号端口。
- › `-v /root/data:/data` 选项将主机的 `/root/data` 目录连接到容器的 `/data` 目录。若将文件放入 `/root/data` 目录, 则能从容器读取相应文件。

显示正在运行的容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3c06a0bebab6	hello:0.1	nginx	10 minutes ago	Up 10 minutes	443/tcp, 0.0.0.0:80->80/tcp	hello-nginx

可以看出, 已经运行了 `hello-nginx` 容器。

运行网页浏览器, 连接到 `http://<主机 IP>:80`, 出现 `Welcome to nginx!` 页面。

第5章

DOCKER

查看 Docker

前面学习了基本命令的用法以及创建镜像的方法，本章将学习如何访问镜像与容器信息、如何从容器中复制文件、如何查看更改文件，以及如何将更改项目存储为镜像。

5.1 使用 history 命令查看镜像历史

下面查看前面创建的 hello:0.1 镜像的历史。

```
$ sudo docker history hello:0.1
```

IMAGE	CREATED	CREATED BY	SIZE
e54ca5efa2e9	Less than a second ago	/bin/sh -c apt-get update && apt-get install	8 B
6c37f792ddac	Less than a second ago	/bin/sh -c apt-get update && apt-get install	83.43 MB
83ff768040a0	Less than a second ago	/bin/sh -c sed -i 's/^#\s*(deb.*universe\)/\$/'	1.903 kB
2f4b4d6a4a06	Less than a second ago	/bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic	194.5 kB
d7ac5e4f1812	Less than a second ago	/bin/sh -c #(nop) ADD file:adc47d03da6bb2418e	192.5 MB
2031ee0736e8	21 minutes ago	/bin/sh -c #(nop) EXPOSE map[443/tcp:{}]	0 B
aa7183744747	21 minutes ago	/bin/sh -c #(nop) EXPOSE map[80/tcp:{}]	0 B
747f1cc74b12	21 minutes ago	/bin/sh -c #(nop) CMD [nginx]	0 B
6103d971baec	21 minutes ago	/bin/sh -c #(nop) WORKDIR /etc/nginx	0 B
ba6879ea73b0	21 minutes ago	/bin/sh -c #(nop) VOLUME ["/data", "/etc/ngin	0 B
9bed2a790c60	21 minutes ago	/bin/sh -c chown -R www-data:www-data /var/li	7 B
d51d99711779	21 minutes ago	/bin/sh -c echo "\ndaemon off;" >> /etc/nginx	1.621 kB
39c72a05ccf4	21 minutes ago	/bin/sh -c apt-get install -y nginx	18.07 MB
c7bc6a6c45a2	25 minutes ago	/bin/sh -c apt-get update	4.206 MB
23f17c89799a	25 minutes ago	/bin/sh -c #(nop) MAINTAINER Foo Bar <foo@bar	0 B
511136ea3c5a	11 months ago		0 B

命令格式为 `docker history< 镜像名称 >:< 标签 >`，也可以使用镜像 ID 代替镜像名称。

依据 Dockerfile 中的设置生成历史。

5.2 ▶ 使用 cp 命令复制文件

下面从 hello-nginx 容器复制文件。

```
$ sudo docker cp hello-nginx:/etc/nginx/nginx.conf ./
```

命令格式为 `docker cp< 容器名称 >:< 路径 >< 主机路径 >`。

执行上述命令，将 `nginx.conf` 文件复制到当前目录。

5.3 ▶ 使用 commit 命令从容器的修改中创建镜像

`docker commit` 命令从容器的修改中创建新的镜像。

假设 hello-nginx 容器中的文件内容发生变化，将容器创建为镜像文件。

```
$ sudo docker commit -a "Foo Bar <foo@bar.com>" -m "add hello.txt" hello-nginx hello:0.2
```

命令格式为 `docker commit < 选项 >< 容器名称 >< 镜像名称 >:< 标签 >`，也可以使用容器 ID 代替容器名称。

`-a "Foo Bar<foo@bar.com>"` 与 `-m "add hello.txt"` 选项用于设置提交的用户与注册信息。将 hello-nginx 容器创建为 `hello:0.2` 镜像。

显示镜像列表。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	e54ca5efa2e9	Less than a second ago	276.1 MB
ubuntu	latest	e54ca5efa2e9	Less than a second ago	276.1 MB
hello	0.2	d2e8c352b303	About a minute ago	298.4 MB
hello	0.1	2031ee0736e8	44 minutes ago	298.4 MB

成功创建 `hello:0.2` 镜像。

5.4 ▶ 使用 diff 命令检查容器文件的修改

`docker diff` 命令显示正在运行的容器中修改文件的列表。比较标准是创建容器的镜像内容。

```
$ sudo docker diff hello-nginx
A /data
C /run
A /run/nginx.pid
C /var/lib/nginx
A /var/lib/nginx/body
A /var/lib/nginx/fastcgi
A /var/lib/nginx/proxy
A /var/lib/nginx/scgi
A /var/lib/nginx/uwsgi
C /etc/nginx
A /etc/nginx/site-enabled
C /dev
C /dev/kmsg
```

命令格式为 `docker diff < 容器名称 >`，也可以使用容器 ID 代替容器名称。A 为添加的文件，C 为修改的文件，D 为删除的文件。

5.5 > 使用 inspect 命令查看详细信息

`docker inspect` 命令显示镜像与容器的详细信息。

```
$ sudo docker inspect hello-nginx
```

```
[[{"Args": [],
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
      "nginx"
    ],
    "CpuShares": 0,
    "Cpuset": "",
    "Domainname": "",
    "Entrypoint": null,
    "Env": [
      "HOME=/",
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "ExposedPorts": {
      "443/tcp": {},
      "80/tcp": {}
    },
    "Hostname": "d4369f661b0a",
    "Image": "hello:0.1",
```

5.3 > 使用 cp 命令复制文件

```

... 省略 ...
},
"Name": "/hello-nginx",
"NetworkSettings": {
  "Bridge": "docker0",
  "Gateway": "172.17.42.1",
  "IPAddress": "172.17.0.40",
  "IPPrefixLen": 16,
  "PortMapping": null,
  "Ports": {
    "443/tcp": null,
    "80/tcp": [
      {
        "HostIp": "0.0.0.0",
        "HostPort": "80"
      }
    ]
  },
  "Path": "nginx",
  "ProcessLabel": "",
  "ResolvConfPath": "/etc/resolv.conf",
  "State": {
    "ExitCode": 0,
    "FinishedAt": "0001-01-01T00:00:00Z",
    "Paused": false,
    "Pid": 19804,
    "Running": true,
    "StartedAt": "2014-06-03T22:25:42.838118718Z"
  }
},
... 省略 ...
}]

```

命令格式为 `docker inspect < 镜像或容器名称 >`，也可以使用镜像 ID 和容器 ID 分别代替镜像名称和容器名称。

5.4 > 使用 diff 命令检查容器的修改

第6章

DOCKER

灵活使用 Docker

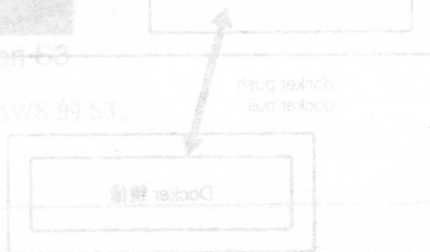


图 6-1 将 Docker 镜像从 Docker Hub 拉取到本地

我们之前学习了 Docker 命令的基本使用方法，本章将进一步学习更多的使用方法。

6.1 ▸ 搭建 Docker 私有仓库

Docker 命令默认使用 Docker Hub。下面创建私有仓库服务器。

Docker 仓库服务器名为 Docker 注册 (registry) 服务器。使用 `docker push` 命令可以将镜像上传到注册服务器，使用 `docker pull` 命令可以下载镜像。

Docker 注册服务器中有多种存储镜像数据的方法，可以存储到执行 Docker 注册的服务器，也可以存储到 Amazon S3。

6.1.1 存储镜像数据到本地

Docker 注册服务器也是 Docker Hub 提供的 Docker 镜像。首先下载 Docker 注册镜像。

```
$ sudo docker pull registry:latest
```

以容器运行 `registry:latest` 镜像。

```
$ sudo docker run -d -p 5000:5000 --name hello-registry \  
-v /tmp/registry:/tmp/registry \  
registry
```

运行后，镜像文件存储到主机的 `/tmp/registry` 目录。

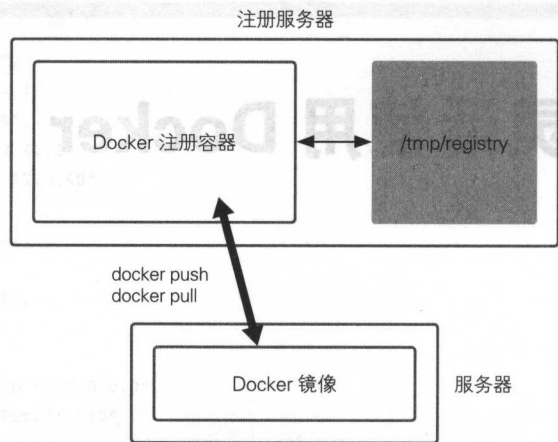


图 6-1 将镜像数据存储到 Docker 注册和本地

6.1.2 使用 push 命令上传镜像

下面将之前创建的 hello:0.1 镜像上传到私有仓库。

```
$ sudo docker tag hello:0.1 localhost:5000/hello:0.1
$ sudo docker push localhost:5000/hello:0.1
```

创建标签的命令格式为 docker tag< 镜像名称 >:< 标签 ><Docker registry URL>/< 镜像名称 >:< 标签 >。

上传镜像的命令格式为 docker push<Docker registry URL>/< 镜像名称 >:< 标签 >。

向私有仓库上传镜像时，需要先创建标签。使用 docker tag 命令为 hello:0.1 镜像创建标签 localhost:5000/hello:0.1，然后使用 docker push 命令将 localhost:5000/hello:0.1 镜像上传到私有仓库（由于已经创建了标签，故实际上传 hello:0.1 镜像）。

接下来，可以从其他服务器连接私有仓库（Docker 注册服务器）并下载镜像。若私有仓库服务器 IP 地址为 192.168.0.39，则执行如下命令。

```
$ sudo docker pull 192.168.0.39:5000/hello:0.1
```

显示镜像列表。

\$ sudo docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos	latest	0c752394b855	4 weeks ago	124.1 MB
192.168.0.39:5000/hello	0.1	2031ee0736e8	4 weeks ago	298.4 MB

从私有仓库下载 192.168.0.39:5000/hello 镜像。

执行如下命令删除镜像。

```
$ sudo docker rmi 192.168.0.39:5000/hello:0.1
```

6.1.3 存储镜像数据到 Amazon S3

下面学习如何将镜像存储到 AWS 的 S3。

首先下载 Docker 注册镜像。

```
$ sudo docker pull registry:latest
```

以容器运行 registry:latest 镜像。

```
$ sudo docker run -d -p 5000:5000 --name s3-registry \
-e SETTINGS_FLAVOR=s3 \
-e AWS_BUCKET=examplebucket10 \
-e STORAGE_PATH=/registry \
-e AWS_KEY=AKIABCDEFGHIJKLMNPOQ \
-e AWS_SECRET=sF4321ABCDEFGHIJKLMNOPqrstuvwxyz21345Afc \
registry
```

若要使用 S3，则使用 -e 选项进行相应设置。

- SETTINGS_FLAVOR: 镜像存储方法，设置为 s3。
- AWS_BUCKET: 存储镜像数据的 S3 bucket 名称。示例中设置为 examplebucket10。各位输入创建的 S3 bucket 名称即可。
- STORAGE_PATH: 镜像数据存储路径。设置为 /registry。
- AWS_KEY: 设置 AWS 访问密钥。
- AWS_SECRET: 设置 AWS 秘密密钥。

接下来，将 Docker 镜像推送 (push) 到 s3-registry 仓库后，镜像数据即存储到 S3 bucket。

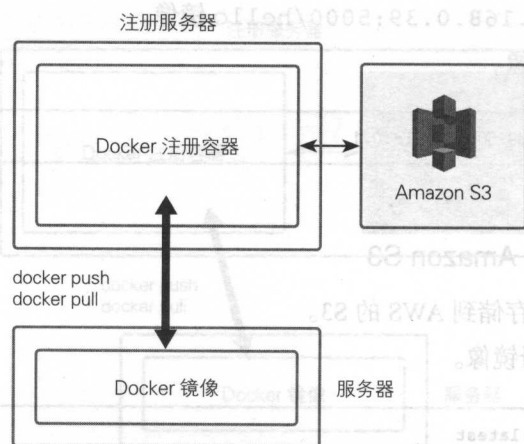


图 6-2 将镜像数据存储在 Docker 注册和 Amazon S3

6.1.4 使用默认认证

Docker 注册中没有登录功能，故必须使用 Nginx 的基本用户验证（Basic Authentication）功能。此外，由于 HTTP 协议不支持认证，故必须使用 HTTPS 协议。

首先编辑 `/etc/hosts` 文件，添加测试域名，修改该文件时必须拥有 root 权限。若未购买域名，则必须设置该部分；若购买域名并设置了 DNS，则该部分可跳过。

> /etc/hosts

```

127.0.0.1    localhost
127.0.1.1    ubuntu
<注册服务器IP地址>    registry.example.com

# The following lines are desirable for IPv6 capable hosts
::1         localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
  
```

请各位将注册服务器的 IP 地址设置为 `registry.example.com`，本书将以 `registry.example.com` 为基准进行说明。

接下来，创建 SSL 自签名（Self Signed）证书。若购买了 SSL 公有证书，则该部分可跳过。执行如下命令，创建私有密钥文件。

```
$ openssl genrsa -out server.key 2048
```

创建证书签名申请（Certificate signing request）文件。

- ▶ Country Name: 国家代码。输入大写字母 CN。
- ▶ State or Province Name: 州或省。根据自身情况输入。
- ▶ Locality Name: 城市。根据自身情况输入。
- ▶ Organization Name: 输入公司名称。
- ▶ Organization Unit Name: 输入组织名称。
- ▶ Common Name: 运行 Docker 注册的服务器域名。若输入不正确, 则使用证书登录时会发生错误。根据 /etc/hosts 文件的设置, 输入 registry.example.com。
- ▶ Email Address: 电子邮件地址。

```
$ openssl req -new -key server.key -out server.csr
```

You are about to be asked to enter information that will be incorporated into your certificate request.

What you are about to enter is what is called a Distinguished Name or a DN.

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

Country Name (2 letter code) [AU]:KO

State or Province Name (full name) [Some-State]:

Locality Name (eg, city) []:Seoul

Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Company

Organizational Unit Name (eg, section) []:Example Company

Common Name (e.g. server FQDN or YOUR name) []:registry.example.com

Email Address []:exampleuser@example.com

Please enter the following 'extra' attributes to be sent with your certificate request

A challenge password []:<不输入任何内容>

An optional company name []:<不输入任何内容>

创建服务器证书文件。

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

下面将 server.crt 证书文件安装到系统 (若不想安装证书文件, 则使用 --insecure-registry 选项。该部分内容将在后文进行说明)。

> Ubuntu

```
$ sudo cp server.crt /usr/share/ca-certificates/
```

```
$ echo "server.crt" | sudo tee -a /etc/ca-certificates.conf
```

```
$ sudo update-ca-certificates
```

> CentOS

```
$ sudo cp server.crt /etc/pki/ca-trust/source/anchors/
$ sudo update-ca-trust enable
$ sudo update-ca-trust extract
```

向 `/etc/hosts` 添加域名，安装证书文件后重启 Docker 服务。只有重启 Docker 服务，所添加的域名与安装证书才能生效。

```
$ sudo service docker restart
```

将 `server.crt` 证书文件复制到要访问 Docker 注册的其他系统，采用相同方式安装，然后重启 Docker 服务。若未购买域名，则在 `/etc/hosts` 中设置注册服务器（`registry.example.com`）的 IP 地址。

提示 --insecure-registry 选项

若不想将 `server.crt` 证书文件安装到系统，那么运行 Docker 守护进程时要使用 `--insecure-registry` 选项。

```
$ sudo docker -d --insecure-registry registry.example.com
```

- 使用 `--insecure-registry` 选项设置 Docker 注册域名。若想设置多个域名，只要重复使用 `--insecure-registry` 选项即可。

我们一般不直接运行 Docker 守护进程，而以服务形式运行。此时要设置 `/etc/init.d/docker` 文件（修改该文件需要拥有 root 权限）的 `DOCKER_OPTS` 部分，如下所示。

> /etc/init.d/docker

```
DOCKER_OPTS="--insecure-registry registry.example.com"
```

`/etc/init.d/docker` 文件修改完成后，重启 Docker 服务。

```
$ sudo service docker restart
```

在要访问 Docker 注册的其他系统中采用相同方式，设置 `--insecure-registry` 选项后运行 Docker 守护进程。

下面创建 `.htpasswd` 文件，该文件存储用户账号与密码。首先安装如下包。

> Ubuntu

```
$ sudo apt-get install apache2-utils
```

> CentOS

```
$ sudo yum install httpd-tools
```

使用 `htpasswd` 命令创建 `.htpasswd` 文件，添加名为 `hellouser` 的用户。在密码输入部分输入要使用的密码。

```
$ htpasswd -c .htpasswd hellouser
New password:<输入密码>
Re-type new password:<输入密码>
Adding password for user hellouser
```

下面不另外创建 Nginx 镜像而使用官方镜像。将如下内容存储为 `nginx.conf` 文件。

➤ `dockerbook/Chapter06/nginx.conf`

➤ `nginx.conf`

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    server {
        listen 443;
        server_name registry.example.com;

        ssl on;
        ssl_certificate /etc/server.crt;
        ssl_certificate_key /etc/server.key;

        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Authorization "";

        client_max_body_size 0;

        chunked_transfer_encoding on;

        location / {
            proxy_pass http://docker-registry:5000;
            proxy_set_header Host $host;
            proxy_read_timeout 900;

            auth_basic "Restricted";
            auth_basic_user_file .htpasswd;
        }
    }
}
```

➤ `server_name`: 设置 Docker 注册服务器的域名。此处设置为 `registry.example.com`。

- `ssl_certificate`、`ssl_certificate_key`: 设置 `/etc/server.crt`、`/etc/server.key`。
- `proxy_pass`: 设置逆向代理。设置为 Docker 注册容器与端口 `docker-registry:5000`。
- `auth_basic`: 设置认证。设置为 `Restricted`, 使用默认认证。
- `auth_basic_user_file`: 设置用于存储用户信息的 `.htpasswd` 文件。

提示 daemon off; 选项

从 Nginx 官方镜像 1.7.5 版本开始, 运行 Nginx 可执行文件时, 在选项中直接设置 `daemon off;`, 如下所示。因此, 也可以不向 `nginx.conf` 文件添加 `daemon off;` 选项。

```
CMD ["nginx", "-g", "daemon off;"]
```

执行如下命令, 首先创建 Docker 注册容器。根据 `nginx.conf` 文件中的设置, 将容器名称设置为 `docker-registry`。

```
$ sudo docker run -d --name docker-registry \
-v /tmp/registry:/tmp/registry \
registry:0.8.1
```

- 由于要通过 Nginx 进行用户验证后收发镜像, 故 Docker 注册的 5000 号端口不暴露在外。

使用 Nginx 官方镜像 1.7.5 版本创建容器, 并与 `docker-registry` 容器连接。

```
$ sudo docker run -d --name nginx-registry \
-v ~/nginx.conf:/etc/nginx/nginx.conf \
-v ~/.htpasswd:/etc/nginx/.htpasswd \
-v ~/server.key:/etc/server.key \
-v ~/server.crt:/etc/server.crt \
--link docker-registry:docker-registry \
-p 443:443 \
nginx:1.7.5
```

- 使用 `-v` 选项将 `nginx.conf` 文件连接至容器的 `/etc/nginx/nginx.conf`。同样, 将 `.htpasswd` 文件连接到 `/etc/nginx/.htpasswd`。就像前面设置的一样, 将 `server.key`、`server.crt` 文件连接到 `/etc` 目录下。

我在用户账户目录下创建设置文件与认证文件, 所以要使用 `~` 符号, 如 `~/nginx.conf`。各位请将创建的设置文件与认证文件路径指定为绝对路径。

- 使用 `--link docker-registry:docker-registry` 选项, 通过 `docker-registry` 别名连接前面创建的 `docker-registry` 容器。这样就可以使用 `nginx`。

conf 的 proxy_pass 设置, 向 Docker 注册发送 traffic。

► 使用 -p 443:443 选项, 将 443 号端口暴露在外。

使用 docker login 命令, 登录 https://registry.example.com。在 Username 与 Password 中输入使用 htpasswd 命令创建的用户名与密码。Email 一项可以不输入。

```
$ sudo docker login https://registry.example.com
Username: hellouser
Password: <输入密码>
Email:
Login Succeeded
```

命令格式为 docker login<Docker registry URL>。

注意 由于繁琐, 所以我们跳过了域名设置, 只使用 IP 地址则无法登录。证书中设置的域名与 docker login 命令中输入的域名必须保持一致。由于 HTTPS 协议不允许访问 IP 地址, 所以要先将未购买的域名注册到 /etc/hosts 文件再使用。

接下来, 将前面创建的 hello:0.1 镜像上传至私有仓库。

```
$ sudo docker tag hello:0.1 registry.example.com/hello:0.1
$ sudo docker push registry.example.com/hello:0.1
The push refers to a repository [registry.example.com/hello] (len: 1)
Sending image list
Pushing repository registry.example.com/hello (1 tags)
511136ea3c5a: Image successfully pushed
bfb8b5a2ad34: Image successfully pushed
c1f3bdbd8355: Image successfully pushed
897578f527ae: Image successfully pushed
9387bcc9826e: Image successfully pushed
809ed259f845: Image successfully pushed
96864a7d2df3: Image successfully pushed
ba3b051655b4: Image successfully pushed
11ae3a0f7f28: Image successfully pushed
e75f421fc19c: Image successfully pushed
507149de4094: Image successfully pushed
ce11bd8322f9: Image successfully pushed
bb5b14a7e9b8: Image successfully pushed
548b98b8a152: Image successfully pushed
e376f7a8e74c: Image successfully pushed
e2626c81818f: Image successfully pushed
7a06b68da607: Image successfully pushed
Pushing tag for rev [7a06b68da607] on {https://registry.example.com/v1/repositories/hello/tags/0.1}
```

镜像标签以 <Docker registry URL>/< 镜像名称 >:< 标签 > 格式进行创建。由于已经设置为 registry.example.com, 所以最终标签为 registry.example.com/hello:0.1。

在另一台服务器中运行如下命令, 即可下载 registry.example.com 中存储的镜像。

```
$ sudo docker pull registry.example.com/hello:0.1
```

6.2 ▶ 连接 Docker 的容器

使用 Docker 创建镜像时, 虽然 Web 服务器、DB 等所需程序都可以安装其中, 但通常要将这些程序分别生成镜像。像这样分别生成镜像、创建容器时, 经常要连接相邻的容器。比如, Web 服务器必须连接 DB 进行数据交换。

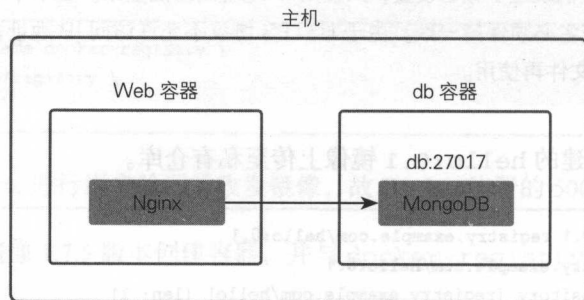


图 6-3 连接 Docker 容器

连接这些 Docker 容器时, 要在 docker run 命令中使用 --link 选项。首先以容器运行 DB 镜像, 此处要使用 MongoDB。(使用 docker run 命令时, 若本地无镜像, 则自动下载镜像。)

```
$ sudo docker run --name db -d mongo
```

将 DB 容器名称设置为 db。

接下来创建 web 容器, 并与 db 容器连接。使用 nginx 镜像创建要用作 Web 服务器的容器。

```
$ sudo docker run --name web -d -p 80:80 --link db:db nginx
```

docker run 命令中, 连接选项的格式为 --link< 容器名称 >:< 别名 >。

输出容器列表。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3971618834cd	nginx:latest	nginx	About a min	Up About a m	0.0.0.0:80->80/tcp	web
8d4031106c57	mongo:2.6	/usr/src/mongo/docke	4 minutes a	Up 4 minutes	27017/tcp	db, web/db

db 容器与 web 容器连接。NAMES 栏中显示为 web/db，表明可以从 web 容器连接到 db 容器。

可以在 web 容器内使用 db:27017 地址连接到 db 容器的 MongoDB。

```
mongodb://db:27017/exampledb
```

从一个容器内连接另一个容器时，所用格式为 < 别名 >:< 端口号 >。

提示 别名与 IP 地址

如下所示，使用 `docker inspect` 命令从 web 容器的详细信息中获取 `hosts` 文件路径后，再使用 `cat` 命令查看其内容（``` 是 TAB 键上方按键上的字符）。

```
$ cat `sudo docker inspect -f "{{ .HostsPath }}" web`
172.17.0.13      aal982fed33e
127.0.0.1       localhost
::1            localhost ip6-localhost ip6-loopback
fe00::0        ip6-localnet
ff00::0        ip6-mcastprefix
ff02::1        ip6-allnodes
ff02::2        ip6-allrouters
172.17.0.12     db
```

db 是 `--link db:db` 中设置的别名，172.17.0.12 是 db 容器的 IP 地址。

连接容器时，IP 地址自动设置到 `hosts` 文件，所以借助别名即可连接到要连接的容器。

6.3 > 连接到其他服务器的 Docker 容器

如前所述，`--link` 选项用于连接同一服务器的多个容器。下面学习使用 Ambassador 容器连接位于不同服务器的容器。

Ambassador 容器不是什么特别的容器，只是普通的 Docker 容器。Ambassador 容器使用 `socat` 程序将 TCP 连接传递到其他地方。

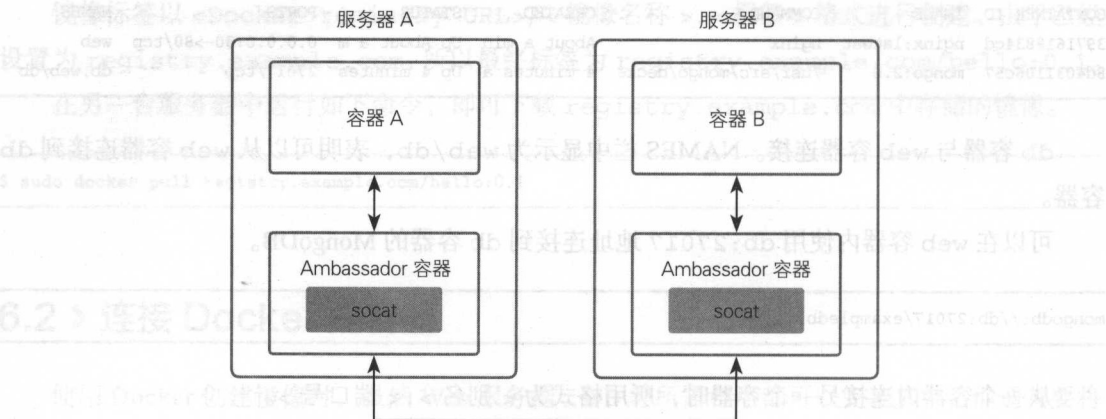


图 6-4 Ambassador 容器的基本概念

查看 Ambassador 容器的 Dockerfile 文件，它看似相当复杂，但比想得简单。这段 shell 脚本在运行 `docker run` 命令时利用传递的环境变量执行 socat。

```
CMD env | grep _TCP= | \
sed 's/.*_PORT_\([0-9]*\) _TCP=tcp://\(\.*\):(.*)/socat \
TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' \
| sh && top
```

在 `docker run` 命令中使用 `--link` 选项或者设置 `-e EXAMPLE_PORT_1234_TCP=tcp://192.168.1.10:1234`，在环境变量中设置如下端口信息。

使用 `env` 命令输出环境变量，使用 `grep` 命令查看包含 `_TCP=` 的字符串。然后通过 `sed` 命令使用正则表达式，从字符串提取端口号与 IP 地址。之后，利用提取的端口号与 IP 地址运行 `socat` 命令。

```
EXAMPLE_PORT=tcp://192.168.0.10:1234
EXAMPLE_PORT_1234_TCP_ADDR=192.168.0.10
EXAMPLE_NAME=/example_ambassador/example
HOSTNAME=0cf479687cb0
EXAMPLE_PORT_1234_TCP_PORT=1234
HOME=/
EXAMPLE_PORT_1234_TCP_PROTO=tcp
EXAMPLE_PORT_1234_TCP=tcp://192.168.0.10:1234
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

上述示例环境中，使用 `socat` 命令将本地 TCP 协议 1234 端口的数据传递到 192.168.0.10 的 1234 端口。这种结构称为 Ambassador Pattern。

从其他服务器的容器中暴露端口后再连接到相应端口会更简单，那么为什么使用这么复杂的方法呢？若在容器中暴露端口，则必须知道相应服务器的 IP 地址或域名。这样就需要在所编程序的源代码中设置 IP 地址或域名。若服务器的 IP 地址、域名发生变化，则必须修改源代码。使用 Ambassador 容器时，由于可以使用别名进行访问，所以不必修改源代码。也就是说，即使是外部服务器，也可以像在相同服务器的 Docker 内部网一样进行访问。

提示 socat 程序

socat 含义为 Socket CAT，它将套接字通信传递给其他通道。通道有文件、管道、设备（串口、pseudo 终端等）、套接字（Unix 套接字、TCP、UDP 等）。

> <http://www.dest-unreach.org/socat/>

接下来，使用 Ambassador 容器连接到其他服务器的容器。为了测试顺利，我们将使用 Redis 数据库。

首先，在用作 Redis 服务器的电脑中创建 Redis 容器，该服务器的 IP 地址为 192.168.0.10。

```
$ sudo docker pull redis:latest
$ sudo docker run -d --name redis redis:latest
```

使用 `--name redis` 选项，将容器名称设置为 `redis`。

接着，为 Redis 容器创建 Ambassador 容器。`svendowideit` 是 Ambassador Pattern 创建者的名字。

```
$ sudo docker run -d --link redis:redis --name redis_ambassador \
-p 6379:6379 svendowideit/ambassador
```

- 使用 `-d` 选项在后台运行容器。
- 使用 `--link redis:redis` 选项，用 `redis` 别名连接 `redis` 容器。
- 使用 `--name redis_ambassador` 选项，将容器名称设置为 `redis_ambassador`。
- 使用 `-p 6379:6379` 选项连接容器的 6379 端口与主机的 6379 端口，并将之暴露在外部。
- 从 Docker Hub 下载 `svendowideit/ambassador` 镜像，然后创建为容器（使用 `docker run` 命令时，若本地无镜像，则自动下载镜像）。

接下来，在要用作 Redis 客户端的电脑中创建 `ambassador` 容器。Redis 服务器的 IP 地址为 192.168.0.10。

```
$ sudo docker run -d --name redis_ambassador --expose 6379 \
-e REDIS_PORT_6379_TCP=tcp://192.168.0.10:6379 svendowideit/ambassador
```

- ▶ 使用 `-d` 选项在后台运行容器。
- ▶ 使用 `--name redis_ambassador` 选项，设置容器名称为 `redis_ambassador`。
- ▶ 使用 `--expose 6379` 选项，使得可以从其他容器连接到 6379 端口。也就是说，Redis 客户端连接到该 `redis_ambassador` 容器的 6379 端口。与 `-p` 选项不同，`--expose` 选项不会将主机端口暴露到外面。
- ▶ 使用 `-e REDIS_PORT_6379_TCP=tcp://192.168.1.10:6379` 选项设置 IP 地址与端口，连接到位于其他服务器的 `redis_ambassador` 容器。
- ▶ 从 Docker Hub 下载 `svendowideit/ambassador` 镜像，然后以容器进行创建。

Docker Hub 中也有只包含 Redis 客户端的容器，使用该容器连接到 `redis_ambassador` 容器。

```
$ sudo docker run -i -t --rm --link redis_ambassador:redis relateiq/redis-cli
```

- ▶ 使用 `-i -t` 选项，使得可以在控制台中进行输入输出。
- ▶ 使用 `--rm` 选项只运行容器，删除容器本身。与 Redis 客户端一样，一次性使用时会比较方便。
- ▶ 使用 `--link redis_ambassador:redis` 选项，通过 `redis` 别名连接 `redis_ambassador` 容器。
- ▶ 从 Docker Hub 下载 `relateiq/redis-cli` 镜像，然后以容器进行创建。

运行 Redis 客户端后，输入 `ping` 命令。若命令执行结果为 `PONG`，则表明正常连接至其他服务器的 Redis 容器。

```
redis 172.17.0.4:6379> ping
PONG
redis 172.17.0.4:6379>
```

6.4 ▶ 使用 Docker 数据卷

Docker 数据卷将数据存储到主机而非容器，多个容器需要共享数据时，常常使用数据卷。

Docker 容器中的文件更改项由 Union File System 进行管理，但数据卷不通过 Union File System 直接存储到主机。即使借助 `docker commit` 命令从容器的修改中创建新镜像，数据卷的修改项也不会包含到所创的镜像之中。

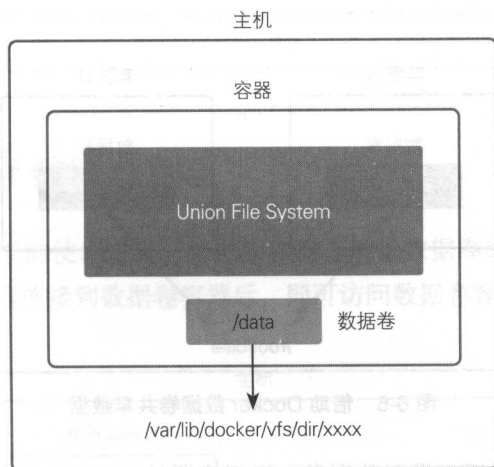


图 6-5 Docker 数据卷

使用如下命令，可以将容器内的 /data 目录设置为数据卷。运行容器的 Bash shell 进入 /data 目录后，创建名为 hello 的空文件。然后执行 exit 命令，退出 Bash shell。

```
$ sudo docker run -i -t --name hello-volume -v /data ubuntu /bin/bash
root@019265a6920f:/# cd /data
root@019265a6920f:/data# touch hello
root@019265a6920f:/data# exit
```

数据卷的选项格式为 -v < 容器目录 >。

使用 docker inspect 命令，查看 hello-volume 容器的数据卷路径。

```
$ sudo docker inspect -f "{{ .Volumes }}" hello-volume
map[/data:/var/lib/docker/vfs/dir/0e3cacb43f11d42b4a6f186198e3e2c812a8ff62c0ab7472d18e1a9735093ae1]
```

使用 ls 命令输出上面所查目录 (/var/lib/docker/vfs/dir/xxxx) 内的文件列表。每次创建容器时，也会同时创建该目录。

```
$ sudo ls /var/lib/docker/vfs/dir/0e3cacb43f11d42b4a6f186198e3e2c812a8ff62c0ab7472d18e1a9735093ae1hello
```

从命令执行结果可以看到前面创建的 hello 文件。若在该目录中创建文件，则在容器内部也可以使用。

接下来，使用数据卷在各个容器间共享数据。

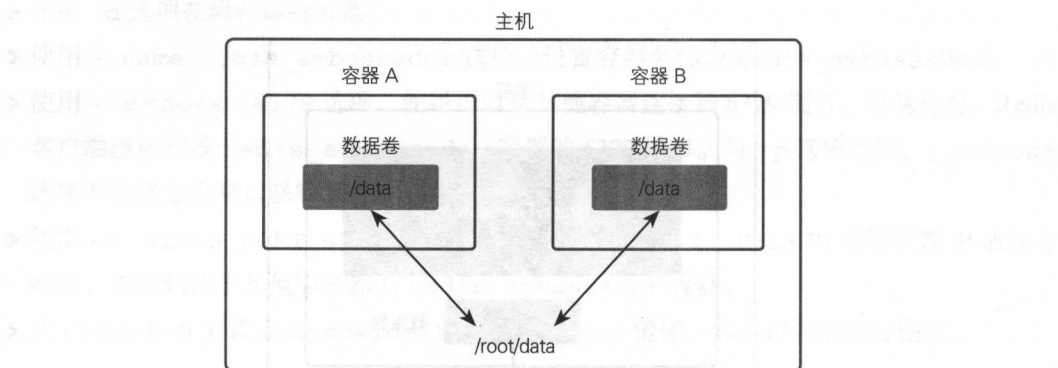


图 6-6 借助 Docker 数据卷共享数据

执行如下命令创建容器后，设置数据卷。运行容器的 Bash shell 转到 `/data` 目录，创建名为 `world` 的空文件，然后执行 `exit` 命令退出 Bash shell。

```
$ sudo docker run -i -t --name hello-volume1 -v /root/data:/data ubuntu /bin/bash
root@f7baf3abefee:/# cd /data
root@f7baf3abefee:/data# touch world
root@f7baf3abefee:/data# exit
```

数据卷的选项格式为 `-v < 主机目录 >:< 容器目录 >`。此处将主机的 `/root/data` 目录连接到 Docker 容器的 `/data` 目录。

输出 `/root/data` 目录的文件列表。

```
$ sudo ls /root/data
world
```

从执行结果可以看到前面创建的 `world` 文件。

下面创建第二个容器。运行容器的 Bash shell，输出 `/data` 目录的文件列表。

```
$ sudo docker run -i -t --name hello-volume2 -v /root/data:/data ubuntu /bin/bash
root@af5a7bdb3e5a:/# ls /data
world
```

从执行结果看，前面创建的 `world` 文件也出现在 `hello-volume2` 文件中。若在 `/data` 目录中创建文件，则在主机与 `hello-volume1` 容器也可以使用。像这样通过数据卷设置，即可实现多个容器共享数据。

除目录外，也可以只将主机的一个文件连接到容器。

```
$ sudo docker run -i -t --name hello-volume -v /root/hello.txt:/root/hello.txt \
ubuntu /bin/bash
```

6.5 > 使用 Docker 数据卷容器

我们前面学习了数据卷的使用方法。数据卷容器是设置数据卷的容器，专门提供数据卷供其他容器共享。从普通容器连接到数据卷容器后，即可访问数据卷容器内的数据卷目录。

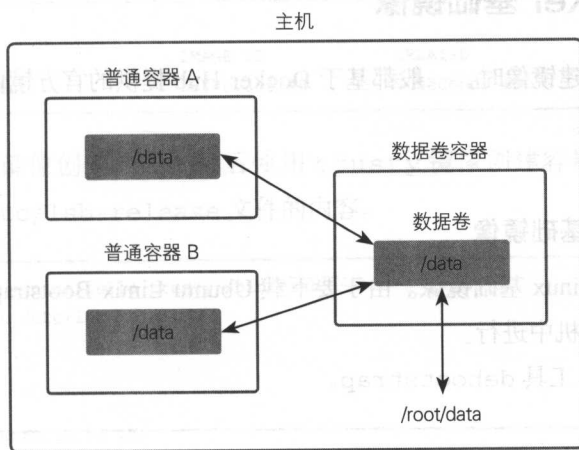


图 6-7 Docker 数据卷容器

输入如下命令，创建数据卷容器。（若容器名重复，则使用 `docker rm` 命令将原有容器删除。）运行容器的 Bash shell 转到 `/data` 目录，然后创建名为 `hello2` 的空文件。依次按 `Ctrl+P`、`Ctrl+Q`，在不停止容器的前提下退出 Bash shell。

```
$ sudo docker run -i -t --name hello-volume -v /root/data:/data ubuntu /bin/bash
root@c9779e329513:/# cd /data
root@c9779e329513:/data# touch hello2
```

创建普通容器，连接到刚刚创建的 `hello-volume` 数据卷容器。运行容器的 Bash shell，输出 `/data` 目录下的文件列表。

```
$ sudo docker run -i -t --volumes-from hello-volume --name hello ubuntu /bin/bash
root@c85aaf93b14e:/# ls /data
hello2
```

用于连接数据卷容器的选项格式为 `--volume-from <数据卷容器>`。

现在在数据卷容器中可以看到创建的 `hello2` 文件。（由于已经连接到主机的 `/root/data`，故也能看到前面创建的其他文件。）

现在虽然只连接了一个普通容器，但也可以将多个普通容器连接到数据卷容器。

如下命令所示，即使不将 `/data` 目录连接到主机的特定目录，也可以用作数据卷容器。

```
$ sudo docker run -i -t --name hello-volume -v /data ubuntu /bin/bash
```

6.6 ▶ 创建 Docker 基础镜像

通过 `Dockerfile` 创建镜像时，一般都基于 Docker Hub 提供的官方镜像。本节将学习创建个人私有基础镜像的方法。

6.6.1 创建 Ubuntu 基础镜像

下面创建 Ubuntu Linux 基础镜像。由于要下载 Ubuntu Linux Bootstrap 二进制文件，所以在安装 Ubuntu Linux 的主机中进行。

首先安装 Bootstrap 工具 `debootstrap`。

```
$ sudo apt-get install debootstrap
```

使用 `debootstrap` 下载 `Ubuntu trusty (14.04)` 二进制文件。`trusty` 是 Ubuntu Linux 的代号。

```
$ sudo debootstrap trusty trusty
```

命令格式为 `debootstrap < 代号 > < 目录 >`。

提示 ▶ Ubuntu Linux 的代号

每个 Ubuntu Linux 版本的代码都可以从如下 URL 中查看。

> <https://wiki.ubuntu.com/DevelopmentCodeNames>

下载全部二进制文件后，使用 `docker import` 命令创建基础镜像。

```
$ sudo tar -C trusty -c . | sudo docker import - trusty
```

`tar -C trusty -c` 命令用于将 `trusty` 目录下的内容压缩为一个文件，并输出至 `stdout`。通过 `|`（管道）可以将输出内容传递给 `docker import` 命令。

命令格式为 `docker import <URL 或 -> <镜像名称>:<标签>`。如下所示，可以使用网络上的文件。若通过 `|` 接收数据，则指定为 `-`。

```
$ sudo docker import http://example.com/trusty.tgz trusty
```

输出镜像目录。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
trusty	latest	5857e0393148	5 seconds ago	228.3 MB

至此，`trusty` 镜像创建成功，然后使用 `trusty` 镜像创建容器。若运行容器的 `Bash` shell，则可以查看 `/etc/lsb-release` 文件的内容。

```
$ sudo docker run -i -t --name hello trusty /bin/bash
root@158ea15ee10c:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04 LTS"
```

代号为 `trusty`，部署版本为 `14.04`。

6.6.2 创建 CentOS 基础镜像

下面创建 CentOS 基础镜像。由于要下载用于 CentOS 的 Bootstrap 二进制文件，所以在安装有 CentOS 的主机中进行。此处以发行版本 CentOS 6.5 为基准。

首先，安装 Bootstrap 工具 `febootstrap`。

```
$ sudo yum install febootstrap
```

使用 `febootstrap` 工具下载 CentOS 6.5 二进制文件。

```
$ sudo febootstrap -u http://ftp.kaist.ac.kr/CentOS/6.5/updates/x86_64/ \
centos65 centos65 http://ftp.kaist.ac.kr/CentOS/6.5/os/x86_64/
```

命令格式为 `febootstrap<选项><仓库><目录><镜像URL>`。

下载全部二进制文件后，使用 `docker import` 命令创建基础镜像。

```
$ sudo tar -C centos65 -c . | sudo docker import - centos65
```

tar -C centos65 -c . 命令用于将 centos65 目录下的内容压缩为一个文件，并输出至 stdout。通过 |（管道）可以将输出内容传递给 docker import 命令。

命令格式为 docker import <tar 文件 URL 或 -> <镜像名称>:<标签>。如下所示，可以使用网络上的文件。若通过 | 接收数据，则指定为 -。

```
$ sudo docker import http://example.com/centos65.tgz centos65
```

输出镜像目录。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos65	latest	8da697bd579e	8 minutes ago	429.9 MB

至此，centos65 镜像创建成功，然后使用 centos65 镜像创建容器。若运行容器的 Bash shell，则可以查看 /etc/centos-release 文件的内容。

```
$ sudo docker run -i -t --name hello centos65 /bin/bash
```

```
bash-4.1# cat /etc/centos-release
```

```
CentOS release 6.5 (Final)
```

发行版本显示为 CentOS release6.5 (Final)。

6.6.3 创建空基础镜像

下面学习如何创建不包含任何内容的空基础镜像。Docker 中，将空的基础镜像称为 scratch 镜像。

使用 /dev/null 设备创建空 tar 文件，并传递给 docker import 命令。

```
$ tar cv --files-from /dev/null | sudo docker import - scratch
```

由于 scratch 镜像不包含任何内容，故不能以容器创建。此处只要编写 Dockerfile 文件并将创建好的可执行文件放入其中即可。

接下来，将使用 C 语言编写的简单程序放入 scratch 镜像，创建 hello 目录并转到此处。

```
$ mkdir hello
```

```
$ cd hello
```

将如下内容保存为 `hello.c`。

> hello.c

```
#include <stdio.h>

int main ()
{
    printf("Hello Docker\n");
    return 0;
}
```

编译 `hello.c` 文件，生成可执行文件。由于 `scratch` 镜像不含任何库文件，所以必须编译为静态二进制文件。

```
~/hello$ gcc hello.c -static -o hello
```

提示 安装编译器

> Ubuntu

```
$ sudo apt-get install gcc
```

> CentOS

```
$ sudo yum install gcc
$ sudo yum install glibc-static
```

将以下内容保存为 `Dockerfile` 文件。

> Dockerfile

```
FROM scratch
ADD ./hello /hello
CMD ["/hello"]
```

以 `scratch` 镜像为基础创建新镜像。

- **FROM**：指定以哪个镜像为基础。Docker 镜像基于已经创建的镜像，格式为 `< 镜像名称 >: < 标签 >`。此处设置为前面已经创建的 `scratch` 镜像。
- **ADD**：设置镜像中要包含的文件，格式为 `< 本地路径 > < 镜像路径 >`。此处设置为前面编译 `hello.c` 文件生成的 `hello` 文件。
- **CMD**：容器启动时要运行的可执行文件或脚本。此处设置为 `hello` 文件。

使用 `docker build` 命令创建镜像。

```
~/hello$ sudo docker build --tag hello:0.1 .
```

然后，以容器创建 `scratch` 镜像创建的 `hello:0.1` 镜像。

```
$ sudo docker run --rm hello:0.1
Hello Docker
```

输出 `Hello Docker`，表明可执行文件正常运行。

6.7 在 Docker 内运行 Docker

本节学习如何在 Docker 容器内运行 Docker。为什么要如此麻烦地在 Docker 容器内运行 Docker 呢？比如，利用 Jenkins 或 CruiseControl 这类自动化构建系统创建 Docker 镜像时，若将 Jenkins 或 CruiseControl 环境本身也创建为 Docker 镜像，则必须能够在 Docker 容器内运行 Docker。

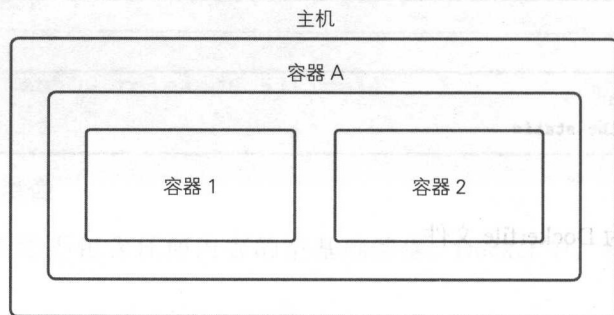


图 6-8 Docker in Docker

首先从 GitHub 下载 Dockerfile 与 Bash 脚本。

```
$ git clone https://github.com/pyrasis/dind.git
```

转到 `dind` 目录后，使用 `docker build` 命令创建镜像。

```
~$ cd dind
~/dind$ sudo docker build --tag dind .
```


短暂等待后创建镜像。执行如下命令，使用 dind 镜像创建容器。

```
~/dind$ sudo docker run -i -t --privileged dind
root@ee112b504b98:/# 2014/08/16 17:20:23 docker daemon: 1.1.2 d84a070; execdriver: native; graphdriver:
[48756c49] +job initserver()
[48756c49.initserver()] Creating server
[48756c49] +job serveapi(unix:///var/run/docker.sock)
2014/08/16 17:20:23 Listening for HTTP on unix (/var/run/docker.sock)
[48756c49] +job init_networkdriver()
[48756c49.init_networkdriver()] creating new bridge for docker0
[48756c49.init_networkdriver()] getting iface addr
[48756c49] -job init_networkdriver() = OK (0)
2014/08/16 17:20:23 WARNING: Your kernel does not support cgroup swap limit.
Loading containers: : done.
[48756c49.initserver()] Creating pidfile
[48756c49.initserver()] Setting up signal traps
[48756c49] -job initserver() = OK (0)
[48756c49] +job acceptconnections()
[48756c49] -job acceptconnections() = OK (0)

root@ee112b504b98:/#
```

此处的 `--privileged` 选项很重要，它使得在容器内部可以使用主机的所有 Linux 内核功能。

Docker in Docker 是实验性质的功能，所以要设置日志输出。若不想输出日志，只需使用 `-e LOG=file` 选项，如下所示。

```
$ sudo docker run -i -t --privileged -e LOG=file dind
```

接下来，在 Docker 容器内运行 Docker。输入如下命令，运行 busybox。

```
root@ee112b504b98:/# sudo docker run -i -t busybox:latest /bin/sh
Unable to find image 'busybox:latest' locally
Pulling repository busybox
a9eb17255234: Download complete
511136ea3c5a: Download complete
42eed7f1bf2a: Download complete
120e218dd395: Download complete
/ #
```

像这样，依照主机→dind 容器→busybox 容器顺序执行。

提示 Docker in Docker 与 Linux 发行版

Docker in Docker 受主机 Linux 内核版本影响，所以在不同的 Linux 发行版下，它有可能无法正常工作。

- Ubuntu 14.04: 正常运行。
- CentOS 6.5: 内核版本低，发生内核错误。
- CentOS 7: 经常发生回环设备挂载失败。

建议使用的 Docker 版本为 1.1.2 以上，在 1.0.x 版本中不运行。

6.7 在 Docker 内运行 Docker

本章将介绍如何在 Docker 容器内运行 Docker，即 Docker in Docker (DinD)。Docker in Docker 允许你在 Docker 容器内运行 Docker，从而可以在容器内运行 Docker 容器。这在某些场景下非常有用，例如在容器内运行 Docker 容器，以便在容器内运行 Docker 容器。Docker in Docker 的实现方式有多种，本章将介绍其中一种实现方式。Docker in Docker 的实现方式有多种，本章将介绍其中一种实现方式。

在 Docker 容器内运行 Docker 容器，需要满足一些条件。首先，宿主机的 Linux 内核版本必须足够高，以支持 Docker in Docker。其次，宿主机的 Docker 版本必须足够高，以支持 Docker in Docker。最后，宿主机的 Docker 配置必须正确，以支持 Docker in Docker。Docker in Docker 的实现方式有多种，本章将介绍其中一种实现方式。

在 Docker 容器内运行 Docker 容器，需要满足一些条件。首先，宿主机的 Linux 内核版本必须足够高，以支持 Docker in Docker。其次，宿主机的 Docker 版本必须足够高，以支持 Docker in Docker。最后，宿主机的 Docker 配置必须正确，以支持 Docker in Docker。Docker in Docker 的实现方式有多种，本章将介绍其中一种实现方式。

第7章

DOCKER

详细了解 Dockerfile

我们在第4章学习镜像创建时，简单介绍了编写 Dockerfile 的方法。本章将进一步了解有关 Dockerfile 的内容。

Dockerfile 的编写格式为 < 命令 >< 形式参数 >，如下所示。# 是注释，命令不区分大小写，一般使用大写字母。

```
# 注释
FROM scratch
```

Docker 会依据 Dockerfile 文件中编写的命令顺序依次执行命令。Dockerfile 文件中，命令总是以 FROM 开始。若无 FROM 命令或者在 FROM 之前有其他命令，则无法创建镜像。此外，各命令是独立运行的。比如，即便使用 RUN cd /home/hello 转移目录，也不会对后面的命令产生影响。

创建镜像时，要在 Dockerfile 所在的目录使用 docker build 命令。

```
$ sudo docker build --tag example .
$ sudo docker build --tag pyrasis/example .
```

--tag 或 -t 选项用于设置镜像名称。若想将镜像上传到 Docker Hub，只要在 / 之前添加用户名即可，比如 pyrasis/example。

即使不设置镜像名称也能创建镜像，需要使用该镜像时，只要指定其 ID 即可。

7.1 > .dockerignore

所有位于 Dockerfile 目录下的文件都称为“上下文”(context)。特别是在创建镜像时，由于所有上下文都会传送到 Docker 守护进程，所以请各位不要将非必要的文件放到该目录。

需要从上下文中忽略某些文件或目录时，只要使用 .dockerignore 文件即可。Docker 是采用 Go 语言编写的，文件匹配也遵循 Go 语言规则。

> <http://golang.org/pkg/path/filepath/#Match>

> .dockerignore

```
example/hello.txt
example/*.cpp
wo*
*.cpp
.git
.svn
```

我们既可以忽略指定的文件或路径，也可以使用 * 通配符忽略一批文件。使用版本管理系统管理 Dockerfile 所需文件时，需要忽略 .git、.svn 这类目录。

7.2 > FROM

FROM 用于设置以哪种镜像为基础创建镜像。由于使用 Dockerfile 创建镜像时总以已有镜像为基础，所以必须使用 FROM 命令。

如下使用时，既可以只设置镜像名称，也可以同时设置镜像名称与标签。若只设置镜像名称，则模式使用 latest 标签。请注意，镜像名称不可省略。

> Dockerfile

```
FROM ubuntu
```

> Dockerfile

```
FROM ubuntu:14.04
```

命令使用格式为 FROM< 镜像 > 或 FROM< 镜像 >:< 标签 >。

如前所述，Dockerfile 中必须使用 FROM 命令，且必须为第一条命令。创建镜像时，若本地存在 FROM 中指定的镜像，则直接使用，否则从 Docker Hub 下载。

一个 Dockerfile 文件中可以使用多个 FROM 命令，若设置了两个 FROM 命令，则创建两个

镜像。若使用 `--tag` 选项设置了镜像名称，则会应用于最后一个 `FROM` 命令。

7.3 ▶ MAINTAINER

`MAINTAINER` 用于设置镜像创建者的信息，格式自由，一般输入名字与电子邮箱即可，如下所示。

> Dockerfile

```
MAINTAINER Hong, Gildong <gd@yuldo.com>
```

使用格式为 `MAINTAINER< 创建者信息 >`。`MAINTAINER` 也可以省略。

7.4 ▶ RUN

`RUN` 用于在 `FROM` 中设置的镜像上运行脚本或命令。`RUN` 运行结果会生成新的镜像，运行的详细信息记录到镜像历史。

1. 使用 shell (/bin/sh) 运行命令

> Dockerfile

```
RUN apt-get install -y nginx
RUN echo "Hello Docker" > /tmp/hello
RUN curl -sSL https://golang.org/dl/go1.3.1.src.tar.gz | tar -v -C /usr/local -xz
RUN git clone https://github.com/docker/docker.git
```

使用格式为 `RUN< 命令 >`，可以使用 `shell` 脚本语句。使用它可以运行 `FROM` 所设镜像内的 `/bin/sh` 可执行文件，若 `/bin/sh` 可执行文件不存在，则无法使用。

2. 无 shell 直接运行

> Dockerfile

```
RUN ["apt-get", "install", "-y", "nginx"]
RUN ["/user/local/bin/hello", "--help"]
```

使用格式为 `RUN["< 可执行文件 >", "< 形式参数 1>", "< 形式参数 2>"]`。以数组形式设置可执行文件与形式参数，这种方法不使用 `FROM` 所设镜像的 `/bin/sh` 可执行文件。由于

不识别 shell 脚本语法，所以与 shell 脚本语法相关的字符能够直接传递给可执行文件。

RUN 执行结果会被缓存，在下次创建时再次使用。若不想使用缓存结果，只要在 docker build 命令中使用 --no-cache 选项即可。

7.5 > CMD

CMD 用于设置容器启动时运行的脚本或命令，即使用 docker run 命令创建容器或使用 docker start 命令启动停止的容器时运行。Dockerfile 文件中，CMD 只能使用 1 次。

1. 用 shell (/bin/sh) 运行命令

> Dockerfile

```
CMD touch /home/hello/hello.txt
```

使用格式为 CMD< 命令 >，可以使用 shell 脚本语句。使用它可以运行 FROM 所设镜像内的 /bin/sh 可执行文件，若 /bin/sh 可执行文件不存在，则无法使用。

2. 无 shell 直接运行

> Dockerfile

```
CMD ["redis-server"]
```

3. 无 shell 直接运行时设置形式参数

> Dockerfile

```
CMD ["mysqld", "--datadir=/var/lib/mysql", "--user=mysql"]
```

使用格式为 CMD["< 可执行文件 >", "< 形式参数 1>", "< 形式参数 2>"]。以数组形式设置可执行文件与形式参数，这种方法不使用 FROM 所设镜像的 /bin/sh 可执行文件。由于不识别 shell 脚本语法，所以与 shell 脚本语法相关的字符会直接传递给可执行文件。

4. 使用 ENTRYPOINT 时

> Dockerfile

```
ENTRYPOINT ["echo"]
```

```
CMD ["hello"]
```

使用格式为 `CMD ["<形式参数 1>", "<形式参数 2>"]`。形式参数会传递给 ENTRYPOINT 设置的命令并运行。若 Dockerfile 中存在 ENTRYPOINT，则 CMD 只用于将形式参数传递给 ENTRYPOINT。因此，CMD 无法单独运行文件。

如下所示，创建 Dockerfile 文件和容器后，输出 hello。

```
$ sudo docker build --tag example .
$ sudo docker run example
hello
```

7.6 ▶ ENTRYPOINT

ENTRYPOINT 用于设置容器启动时运行的脚本或命令，即使用 `docker run` 命令创建容器或使用 `docker start` 命令启动停止的容器时运行。Dockerfile 文件中，ENTRYPOINT 只能使用 1 次。

1. 用 shell (/bin/sh) 运行命令

> Dockerfile

```
ENTRYPOINT touch /home/hello/hello.txt
```

使用格式为 `ENTRYPOINT <命令>`，可以使用 shell 脚本语句。使用它可以运行 FROM 所设镜像内的 /bin/sh 可执行文件，若 /bin/sh 可执行文件不存在，则无法使用。

2. 无 shell 直接运行

> Dockerfile

```
ENTRYPOINT ["/home/hello/hello.sh"]
```

> Dockerfile

```
ENTRYPOINT ["/home/hello/hello.sh", "--hello=1", "--world=2"]
```

使用格式为 `ENTRYPOINT ["<可执行文件>", "<形式参数 1>", "<形式参数 2>"]`。以数组形式设置可执行文件与形式参数，这种方法不使用 FROM 所设镜像的 /bin/sh 可执行

文件。由于不识别 shell 脚本语法，与 shell 脚本语法相关的字符会直接传递给可执行文件。

虽然 CMD 与 ENTRYPOINT 都可以用于设置容器启动时要运行的命令，但 docker run 命令中，二者运行方式不同。

如下所示，在 Dockerfile 中通过 CMD 使用 echo 命令，输出 hello。

> Dockerfile

```
FROM ubuntu:latest
CMD ["echo", "hello"]
```

使用 docker run < 镜像 > < 可执行文件 > 命令可以创建容器，并在其中运行给定的命令。若 docker run 命令中指定了可执行文件，则忽略 CMD。

```
$ sudo docker build --tag example .
$ sudo docker run example echo world
world
```

从执行结果看，忽略 CMD ["echo", "hello"], docker run 命令中设置的 echo world 运行后输出 world。docker run 命令中设置的 < 可执行文件 > 与 Dockerfile 的 CMD 具有相同功能。

下面介绍 ENTRYPOINT。如下所示，Dockerfile 中，通过 ENTRYPOINT 使用 echo 命令输出 hello。

> Dockerfile

```
FROM ubuntu:latest
ENTRYPOINT ["echo", "hello"]
```

创建 Dockerfile 文件，并使用 docker run 命令运行。若 docker run 命令中设置了要执行的文件，则不会忽略 ENTRYPOINT，要运行的文件设置本身会处理为形式参数。

```
$ sudo docker build --tag example .
$ sudo docker run example echo world
hello echo world
```

ENTRYPOINT ["echo", "hello"] 中运行 echo hello 输出 hello，docker run 命令中设置的内容会处理为 ENTRYPOINT ["echo", "hello"] 的形式参数，与 echo world 一同输出。在 shell 中表示如下。

```
$ echo hello echo world
hello echo world
```

接下来，使用非 echo 命令的其他方式运行。如下所示，若传入 1 2 3 4，则直接输出 1 2 3 4。

```
$ sudo docker run example 1 2 3 4
hello 1 2 3 4
```

ENTRYPOINT 也可以在 docker run 命令中使用 --entrypoint 选项进行设置。使用 --entrypoint 选项运行 cat 命令，输出 /etc/hostname 文件的内容。

```
$ sudo docker run --entrypoint="cat" example /etc/hostname
9efe43ea4d40
```

若设置了 --entrypoint 选项，则忽略 Dockerfile 文件中的 ENTRYPOINT。

7.7 > EXPOSE

EXPOSE 用于设置与主机相连的端口号，与 docker run 命令中的 --expose 选项功能一致。

> Dockerfile

```
EXPOSE 80
EXPOSE 443
```

> Dockerfile

```
EXPOSE 80 443
```

使用格式为 EXPOSE< 端口号 >。使用 1 个 EXPOSE 也可以同时设置多个端口号。

EXPOSE 只用于与主机进行连接，并不对外暴露。若想将端口暴露在外，需要使用 docker run 命令的 -p、-P 选项。

7.8 > ENV

ENV 用于设置环境变量。使用 ENV 设置的环境变量应用于 RUN、CMD、ENTRYPOINT。

> Dockerfile

```
ENV GOPATH /go
```

```
ENV PATH /go/bin:$PATH
```

使用格式为 ENV< 环境变量 >< 值 >。使用环境变量时，要添加 \$ 符号。

下面使用 CMD 输出 ENV 中设置的环境变量。

> Dockerfile

```
ENV HELLO 1234
CMD echo $HELLO
```

创建 Dockerfile，使用 docker run 命令运行。

```
$ sudo docker build --tag example .
$ sudo docker run example
1234
```

从执行结果看，输出 ENV 中设置的 HELLO 值 1234。

环境变量也可以在 docker run 命令中进行设置。

```
$ sudo docker run -e HELLO=4321 example
4321
```

设置格式为 -e < 环境变量 >=< 值 >。-e 选项可以多次使用，与 --env 选项相同。

7.9 > ADD

ADD 用于向镜像添加文件。

> Dockerfile

```
ADD hello-entrypoint.sh /entrypoint.sh
ADD hello-dir /hello-dir
ADD zlib-1.2.8.tar.gz /
ADD hello.zip /
ADD http://example.com/hello.txt /hello.txt
ADD *.txt /root/
```

使用格式为 ADD < 要复制文件的路径 >< 文件在镜像中的路径 >。

> < 要复制文件的路径 > 以上下文目录为基准，不能使用上下文以外的文件、目录或绝对路径。

>> ADD ../hello.txt /home/hello (×)

>> ADD /home/hello/hello.txt /home/hello (×)

► < 要复制文件的路径 > 不仅可以设置为文件，还可以设置为目录。设置为目录时，会复制目录下的所有文件。另外，也可以使用通配符只复制特定文件。

» 例) `ADD *.txt /root/`

► < 要复制文件的路径 > 也可以设置为网络文件的 URL。

» 若 / 在 < 文件在镜像中的路径 > 最后位置，则创建目录，并将文件复制于其下。若设置为 `ADD http://example.com/hello.txt /home/hello/`，则文件会复制到 `/home/hello/` 目录。

► 解压缩并解开 tar 包后添加位于本地的压缩文件（`tar.gz`、`tar.bz2`、`tar.xz`）。但是，对于网络上的压缩文件只进行解压缩，然后添加整个 tar 文件。

» 例) `ADD hello.tar.gz /`（解压缩、解 tar 包后添加）

» 例) `ADD http://zlib.net/zlib-1.2.8.tar.gz /`（只解 gzip 压缩，然后添加 tar 文件。虽然文件内容是 tar，但文件名带有 .gz 后缀，如 `zib-1.2.8.tar.gz`。）

► < 文件在镜像中的路径 > 必须设置为绝对路径。并且，若路径以 / 结尾，则创建目录并将文件复制到该目录。

► 像 `ADD ./ /hello` 一样，添加当前目录时，`.dockerignore` 文件中设置的文件与目录被排除在外。

提示 解压并添加网络文件 URL

如前所述，ADD 只对文件 URL 进行解压缩，而不会将 tar 拆包。此时，只要使用 RUN 和 curl 或 wget 接收文件后解压缩即可。

> Ubuntu, curl

```
FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y curl
RUN curl http://zlib.net/zlib-1.2.8.tar.gz | tar -xz
```

> Ubuntu, wget

```
FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y wget
RUN wget http://zlib.net/zlib-1.2.8.tar.gz -O - | tar -xz
```

> CentOS, curl

```
FROM centos:latest
RUN yum install -y curl tar
RUN curl http://zlib.net/zlib-1.2.8.tar.gz | tar -xz
```


> CentOS, wget

```
FROM centos:latest
RUN yum install -y wget tar
RUN wget http://zlib.net/zlib-1.2.8.tar.gz -O - | tar -xz
```

以下方法用于解压 tar、tar.gz、tar.bz2、tar.xz 文件。

- ubuntu:latest 安装有 bzip2，未安装 xz。要使用 xz 压缩，需要先使用 apt-get 安装 xz-utils 包。
- centos:latest 安装有 xz，未安装 bz2。要使用 bz2，需要先用 yum 安装 bzip2 包。

```
RUN curl http://example.com/hello.tar | tar -x
RUN curl http://example.com/hello.tar.gz | tar -xz
RUN curl http://example.com/hello.tar.bz2 | tar -xj
RUN curl http://example.com/hello.tar.xz | tar -xJ
```

使用 ADD 添加的文件由所有者 (UID) 0、组 (GID) 0 进行设置，权限继承于已有文件的权限。若以 URL 添加，则权限设置为 600。

提示 UID 0、GID 0 为 root 用户。

7.10 > COPY

COPY 用于向镜像添加文件。与 ADD 不同，使用 COPY 添加压缩文件时，不会解压缩，也不能使用文件 URL。

> Dockerfile

```
COPY hello-entrypoint.sh /entrypoint.sh
COPY hello-dir /hello-dir
COPY zlib-1.2.8.tar.gz /zlib-1.2.8.tar.gz
COPY *.txt /root/
```

使用格式为 COPY < 要复制文件的路径 > < 文件在镜像中的路径 >。

> < 要复制文件的路径 > 以上下文目录为基准，不能使用上下文以外的文件、目录或绝对路径。

» 例) COPY ../hello.txt /home/hello(×)

» 例) COPY /home/hello/hello.txt /home/hello(×)

> < 要复制文件的路径 > 不仅可以设置为文件，还可以设置为目录。设置为目录时，会复

制目录下的所有文件。另外，也可以使用通配符只复制特定文件。

» 例) `COPY *.txt /root`

- < 要复制文件的路径 > 不可以设置为网络文件的 URL。
- 压缩文件不会解压缩，直接复制。
- < 文件在镜像中的路径 > 必须设置为绝对路径。并且，若路径以 / 结尾，则创建目录并将文件复制到该目录。
- 像 `COPY ./ /hello` 一样，添加当前目录时，`.dockerignore` 文件中设置的文件与目录会被排除在外。

使用 `COPY` 添加的文件由所有者 (UID) 0、组 (GID) 0 进行设置，权限继承于已有文件的权限。

7.11 > VOLUME

VOLUME 设置用于将目录下的内容存储到主机而非容器，请参考 6.4 节。

> Dockerfile

```
VOLUME /data
VOLUME ["/data", "/var/log/hello"]
```

使用格式为 `VOLUME< 容器目录 >` 或 `VOLUME[" 容器目录 1", " 容器目录 2"]`。既可以直接设置为路径，比如 `/data`；也可以用数组形式进行设置，比如 `["/data", "/var/log/hello"]`。但是，使用 VOLUME 不能与主机的特定目录进行连接。

若想连接数据卷与主机的特定目录，则必须在 `docker run` 命令中使用 `-v` 选项。

```
$ sudo docker run -v /root/data:/data example
```

选项格式为 `-v < 主机目录 >:< 容器目录 >`。

7.12 > USER

USER 用于设置运行命令的用户账号，该用户会应用于 `RUN`、`CMD`、`ENTRYPOINT`。

```
USER nobody
```

使用格式为 USER< 账号用户名 >。

USER 后面的所有 RUN、CMD、ENTRYPOINT 都会得到应用，中间可以设置其他用户以更换用户。

> Dockerfile

```
USER nobody
RUN touch /tmp/hello.txt

USER root
RUN touch /hello.txt
ENTRYPOINT /hello-entrypoint.sh
```

首先以 nobody 用户创建 /tmp/hello.txt 文件，然后用 root 账号创建 hello.txt 文件（只有 root 用户才能在 / 下创建文件），并运行 /hello-entrypoint.sh 文件。

7.13 ▶ WORKDIR

WORKDIR 用于设置执行 RUN、CMD、ENTRYPOINT 命令的目录。

> Dockerfile

```
WORKDIR /var/www
```

使用格式为 WORKDIR< 路径 >。

WORKDIR 设置的目录对其后的所有 RUN、CMD、ENTRYPOINT 都有效，中间设置其他目录可以更换执行目录。

> Dockerfile

```
WORKDIR /root
RUN touch hello.txt

WORKDIR /tmp
RUN touch hello.txt
```

设置 WORKDIR 时也可以用相对路径代替绝对路径。若使用相对路径，先要以设置的 WORKDIR 路径为基准更改目录。最初基准为 /。

> Dockerfile

```
WORKDIR var
WORKDIR www
```

```
RUN touch hello.txt
```

以上代码使用相对路径，从 `/` 转到 `var` 之后，再移动到 `www`。因此，文件最终创建于 `/var/www/ hello.txt`。

7.14 ▶ ONBUILD

将当前镜像作为基础镜像创建其他镜像时，ONBUILD 指令用于设置一些要触发 (trigger) 的操作。ONBUILD 指定的命令在构建镜像时并不执行，而是在其子镜像中执行。换言之，ONBUILD 中定义的指令会在用于生成其他镜像的 Dockerfile 文件的 FROM 指令之后执行。

> Dockerfile

```
ONBUILD RUN touch /hello.txt
ONBUILD ADD world.txt /world.txt
```

指令格式为 ONBUILD <Dockerfile 命令><Dockerfile 命令的形式参数>。除 MAINTAINER、ONBUILD 之外的所有 Dockerfile 指令都可以用于 ONBUILD 指令。

生成某个镜像并以该镜像为基础定制其他镜像时，可以灵活使用 ONBUILD 指令设置触发指令。

如下所示，使用 ONBUILD 指令，设置执行 `RUN touch /hello.txt`。

> Dockerfile

```
FROM ubuntu:latest
ONBUILD RUN touch /hello.txt
```

使用 `docker build` 命令创建 `example` 镜像后，再使用 `docker run` 命令创建容器。容器的 Bash shell 执行后，使用 `ls` 命令列出 `/` 下的文件目录。

```
$ sudo docker build --tag example .
$ sudo docker run -i -t example /bin/bash
root@891ccb6749e9:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

由于使用 ONBUILD 指令进行设置，所以 `example` 镜像中并不会生成 `/hello.txt` 文件。

接下来，使用 FROM 指令，以 `example` 镜像为基础创建新的镜像。

> Dockerfile

```
FROM example
```

使用 `docker build` 命令创建 `example2` 镜像后, 使用 `docker run` 命令创建容器。容器的 Bash shell 执行后, 使用 `ls` 命令列出 / 目录下的文件目录。

```
$ sudo docker build --tag example2 .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM example
# Executing 1 build triggers
Step onbuild-0 : RUN touch /hello.txt
---> Running in 7e77e38db77c
---> 967feb106636
---> 967feb106636
Removing intermediate container 7e77e38db77c
Successfully built 967feb106636
$ sudo docker run -i -t example2 /bin/bash
root@874d3e1fdd6f:~# ls
bin boot dev etc hello.txt home lib lib64 media mnt opt proc root run sbin srv sys tmp
usr var
```

运行 `docker build` 命令时, 显示 `# Executing 1 build triggers` 信息, 接下来执行使用 `ONBUILD` 设置的指令。这样, 通过 `ONBUILD` 在 `example2` 镜像中生成了 `/hello.txt` 文件。

`ONBUILD` 指令仅适用于从当前镜像创建子镜像, 而不适用于“孙子”镜像。也就是说, `ONBUILD` 中定义的指令不会继承。

提示 可以使用 `docker inspect` 命令查看镜像的 `ONBUILD` 指令中定义的内容。

```
$ sudo docker inspect -f "{{ .ContainerConfig.OnBuild }}" example
[RUN touch /hello.txt]
```

第8章

DOCKER

使用 Docker 部署应用程序

我们学习了 Docker 的基本用法与各种功能，本章将学习如何使用 Docker 部署应用程序。

运行服务器和部署应用程序的方法会根据服务环境或系统构建者的不同而不同，本书介绍使用分布式版本管理系统 Git 与 Docker 部署应用程序的方法。

示例程序请从我的 GitHub 下载。

➤ <https://github.com/pyrasis/dockerbook>

8.1 向一台服务器部署应用程序

首先学习如何向一台服务器部署应用程序。

1. 在开发者 PC 中开发应用程序。
2. 使用 `git push` 命令将源代码上传至服务器。
3. 服务器收到向仓库发送的 `git push` 命令后，执行 `git hook`。
4. 在 `git hook` 中创建 Docker 镜像，并以容器运行镜像。

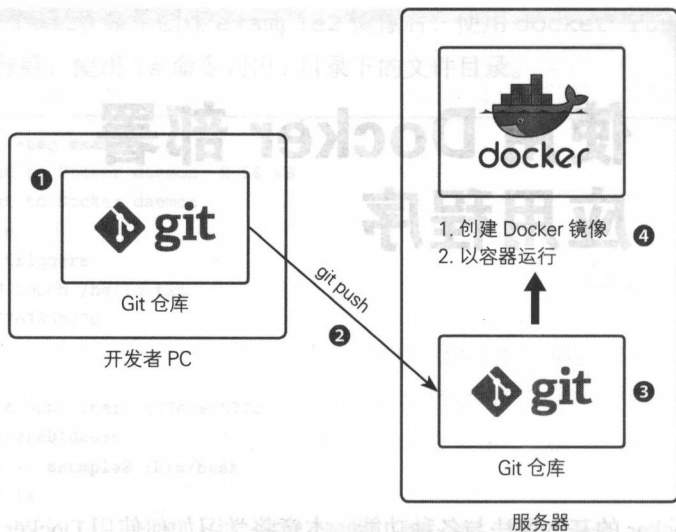


图 8-1 使用 Git 与 Docker 向一台服务器部署应用程序

8.1.1 在开发者 PC 安装 Git 并创建仓库

若开发 PC 中尚未安装 Git，则请先安装（<http://git-scm.com>）。

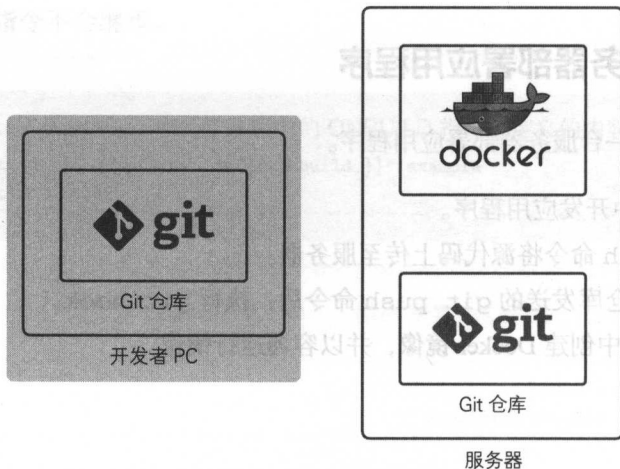


图 8-2 在开发 PC 中安装 Git 并创建代码库

Windows 与 Mac OS X 系统用户只要从以下网址下载并安装文件即可。安装时没有什么需要特别注意的内容，此处不再赘述。

➤ Windows: <http://msysgit.github.com>

➤ Mac OS X: <http://sourceforge.net/projects/git-osx-installer>

> Ubuntu

```
$ sudo apt-get install git
```

> CentOS

```
$ sudo yum install git
```

接下来，在 Windows 中运行 Git Bash。（由于在 Git Bash 中可以运行 Unix/Linux 命令，所以下列命令在 Mac OS X、Linux、Windows 中都是一样的。）在 Mac OS X 中运行终端，在 Linux 中打开终端运行。

创建 Git 仓库，然后转到仓库目录。

```
~$ git init exampleapp
~$ cd exampleapp
```

使用 `git config` 命令设置作者的电子邮件与名称。

```
~/exampleapp$ git config --global user.email gd@yuldo.com
~/exampleapp$ git config --global user.name "Hong, Gildong"
```

8.1.2 在开发者 PC 中使用 Node.js 编写 Web 服务器

下面在开发者 PC 中使用 Node.js 编写简单的 Web 服务器。请将如下内容保存为 `app.js`。

➤ `dockerbook/Chapter08/SingleServerDeployment/exampleapp/app.js`

> ~/exampleapp/app.js

```
var express = require('express');
var app = express();

app.get(['/', '/index.html'], function (req, res) {
  res.send('Hello Docker');
});

app.listen(80);
```

为了使用 Node.js npm 包，编写如下代码，然后保存为 `package.json`。

➤ `dockerbook/Chapter08/SingleServerDeployment/exampleapp/package.json`

```
> ~/exampleapp/package.json
```

```
{
  "name": "exampleapp",
  "description": "Hello Docker",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.x"
  }
}
```

提示 关于 Node.js 的使用方法请参考我编写的文档。

> <http://pyrasis.com/nodejs/nodejs-HOWTO>

使用 `git add`、`git commit` 命令，将文件提交到开发者 PC 的 `exampleapp` 仓库。

```
~/exampleapp$ git add app.js package.json
~/exampleapp$ git commit -m "add source"
```

8.1.3 在开发者 PC 中编写 Dockerfile 文件

下面在开发者 PC 中编写 Dockerfile 文件，以便在服务器中创建 Docker 镜像。请将如下内容保存为 Dockerfile 文件。

➤ `dockerbook/Chapter08/SingleServerDeployment/exampleapp/Dockerfile`

```
> ~/exampleapp/Dockerfile
```

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y nodejs npm
```

```
ADD app.js /var/www/app.js
```

```
ADD package.json /var/www/package.json
```

```
WORKDIR /var/www
```

```
RUN npm install
```

```
CMD nodejs app.js
```

➤ `FROM` 指令用于设置基于 `ubuntu 14.04` 创建镜像。

➤ `RUN` 指令用于安装 `nodejs`、`npm` 包。

- ADD 指令用于将 app.js 与 package.json 复制到镜像的 /var/www 目录。
- WORDIR 指令用于将执行目录更改为 /var/www。用 RUN 指令运行 npm install 命令，将安装 package.json 中设置的 Node.js 模块。
- CMD 设置用于在容器启动时利用 nodejs 运行 app.js（若 Ubuntu 中以包形式安装 Node.js，则运行文件为 nodejs 而非 node）。

将 Dockerfile 文件提交到开发者 PC 的 exampleapp 仓库。

```
~/exampleapp$ git add Dockerfile
~/exampleapp$ git commit -m "add Dockerfile"
```

8.1.4 在开发者 PC 中生成 SSH 密钥

在开发者 PC 中运行 ssh-keygen 命令生成 SSH 密钥（在 Windows 系统下要在 Git Bash 中运行如下命令）。

- Enter file in which to save the key：使用默认值。
- Enter passphrase, Enter same passphrase again：点击 Enter 键，不设置密码。

若 id_rsa、id_rsa.pub 文件已经存在于 /home/< 用户账户 >/.ssh 目录，则跳过该部分。

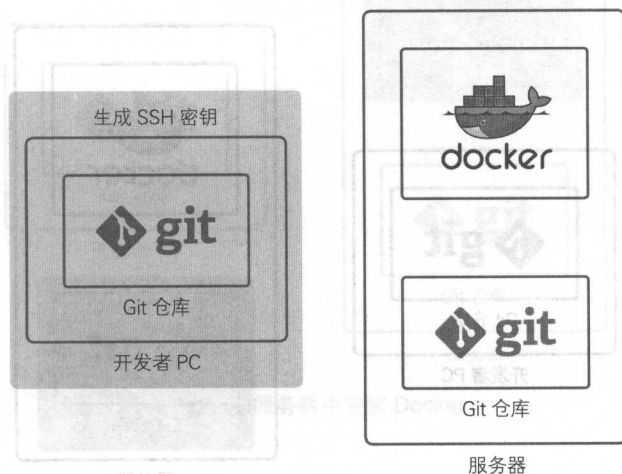


图 8-3 在开发者 PC 中生成 SSH 密钥

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/pyrasis/.ssh/id_rsa):
```

```

Created directory '/home/pyrasis/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pyrasis/.ssh/id_rsa.
Your public key has been saved in /home/pyrasis/.ssh/id_rsa.pub.
The key fingerprint is:
64:9e:5a:8a:0b:93:f0:c5:fb:89:41:31:c4:bb:a2:ab pyrasis@ubuntu
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      ..              |
|      ..              |
|    o. o              |
|   ..o + .            |
|  . +. S              |
| o.+..o +            |
| .+. + o              |
| . o = .              |
|E. o o               |
+-----+

```

可以看到，/home/<用户账户>/.ssh 目录中生成 id_rsa、id_rsa.pub 文件。

8.1.5 在服务器端安装 Git 并创建仓库

下面开始设置服务器。由于 Docker 是 Linux 专用的，所以在 Linux 服务器中进行设置。

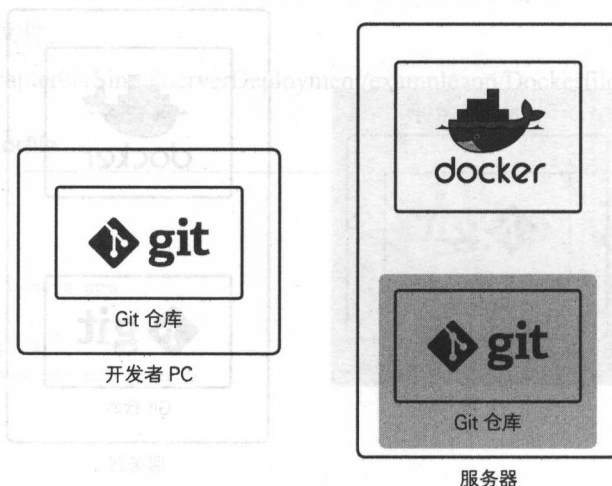


图 8-4 在服务器端安装 Git 并创建仓库

在服务器中安装 Git。

> Ubuntu

```
$ sudo apt-get install git
```

> CentOS

```
$ sudo yum install git
```

在当前 Linux 用户的 home 目录（/home/<用户账户>）创建 exampleapp 仓库，然后将 receive.denycurrentbranch 设置为 ignore，以便从开发者 PC 接收推送的源代码。

```
~$ git init exampleapp
```

```
~$ git config receive.denycurrentbranch ignore
```

8.1.6 在服务器中安装 Docker

下面在服务器中安装要使用的 Docker。关于在 CentOS 中安装 EPEL 的方法，请参考 2.1.3 节。

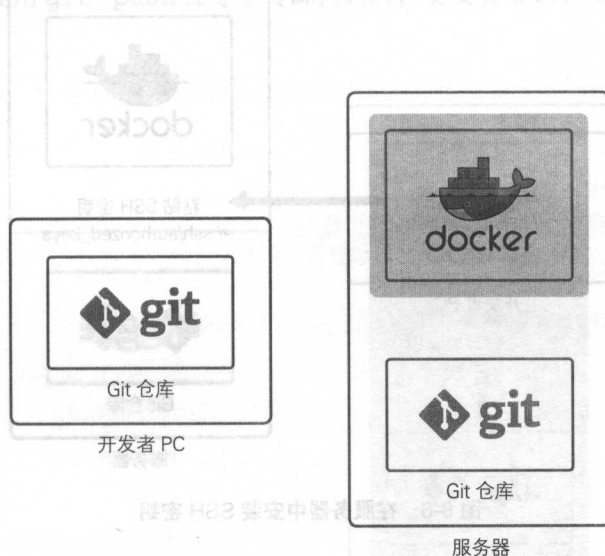


图 8-5 在服务器中安装 Docker

> Ubuntu

```
$ sudo apt-get install docker.io
```

```
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```


> CentOS

```
$ sudo yum install docker-io
$ sudo service docker start
```

使用 `docker` 命令将当前 Linux 用户添加到 `docker` 组，以便能够在不键入 `sudo` 的情形下使用 `root` 权限。

```
$ sudo usermod -aG docker ${USER}
$ sudo service docker restart
```

8.1.7 在服务器中安装 SSH 密钥

下面开始在服务器中安装前面生成的 SSH 密钥，以便开发者 PC 访问服务器时不必使用密码。

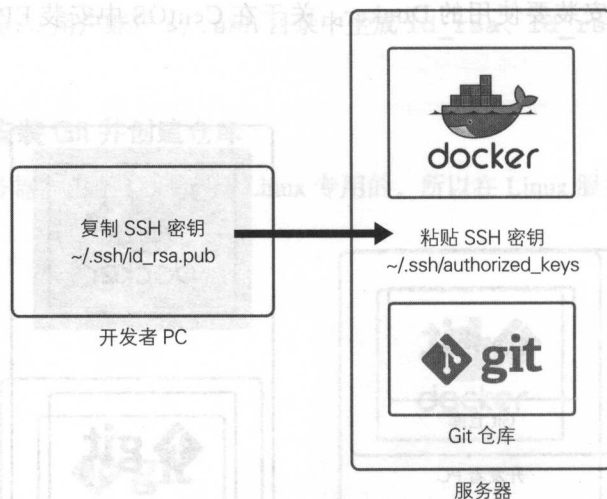


图 8-6 在服务器中安装 SSH 密钥

在服务器的 `/home/<服务器用户账户>` 目录创建 `.ssh` 目录，并设置权限。

```
~$ mkdir .ssh
~$ chmod 700 .ssh
```

在刚刚创建的 `.ssh` 目录创建 `authorized_keys` 文件，然后复制在开发者 PC 中创建的 `id_rsa.pub` 文件的内容，粘贴到 `authorized_keys` 文件。

```
> ~/.ssh/authorized_keys
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQAC4nKkAHdB8gU9DT9te9HDNWA8qTY/9YjgxVr493YxqS3vu+Y5/
UyXgRPMegRZGKWZEDtyssYi/XxQ5R1k0xXF8NE/T/x8DULVc32e3mtA9vnzOMqSHjVLqP0ZZXdhkipEw6s2uKJVtmXQdN+Pz3Dupy
j+la0jww/n3y62goEZ2f9jr7ZnGtL2haYSZJEMh57RVAYwLW3n1Ax53dhKE9ha9xdBC+BRtJkqE8oEq+Vg67H01604kxnRvubiVD
IXfm1/mvXqz+GdgzSta8Uspz59Tth01J0cJbAZyDL5VEBYV5rziHvRmqwDewV0Mn7hZjAuVYDlGPMbW26/hXsdxycMJ pyrasis@
ubuntu
```

以上内容是我的公共密钥。希望各位设置自己的公共密钥（id_rsa.pub）。

为 authorized_keys 文件设置权限。

```
$ chmod 600 authorized_keys
```

这样就可以在开发者 PC 中随意使用 git push 命令而不必输入密码。

8.1.8 在服务器中安装 Git Hook

在开发者 PC 中使用 git push 命令上传源代码时，要安装 Git Hook，以创建 Docker 镜像与容器。

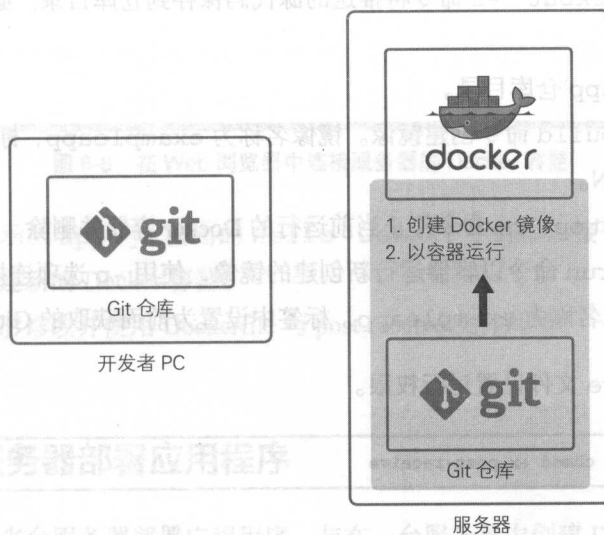


图 8-7 在服务器中安装 Git Hook

在服务器的 /home/<服务器用户账户>/exampleapp/.git/hooks 目录中，将下列内容保存为 post-receive 文件。

```
> dockerbook/Chapter08/SingleDeployment/hooks/post-receive
```

> ~/exampleapp/.git/hooks/post-receive

```
#!/bin/bash

APP_NAME=exampleapp
APP_DIR=$HOME/$APP_NAME
REVISION=$(expr substr $(git rev-parse --verify HEAD) 1 7)

GIT_WORK_TREE=$APP_DIR git checkout -f

cd $APP_DIR
docker build --tag $APP_NAME:$REVISION .
docker stop $APP_NAME
docker rm $APP_NAME
docker run -d --name $APP_NAME -p 80:80 $APP_NAME:$REVISION
```

- APP_NAME 变量用于设置当前应用程序的名称，该名称必须与仓库名称一致。
- APP_DIR 变量用于设置仓库目录路径。HOME 变量中保存着用户的 home 目录路径。（/home/< 用户账户 >）
- 使用 git rev-parse 命令获取推送的最新修订的 REVISION，然后只将前面 7 位数存储到 REVISION 变量。
- 使用 git checkout -f 命令将推送的源代码保存到仓库目录。必须设置 GIT_WORK_TREE 变量。
- 转到 exampleapp 仓库目录。
- 使用 docker build 命令创建镜像。镜像名称为 exampleapp，标签中设置为刚刚获取的 Git REVISION。
- 使用 docker stop、rm 命令停止当前运行的 Docker 容器并删除。
- 使用 docker run 命令以容器运行新创建的镜像，使用 -p 选项连接 80 号端口，并将其暴露在外。镜像名称为 exampleapp，标签中设置为前面获取的 Git REVISION。

为 post-receive 文件设置运行权限。

```
~/exampleapp/.git/hooks$ chmod +x post-receive
```

8.1.9 在开发者 PC 中推送源代码

下面回到开发者 PC。

转到 exampleapp 仓库目录后，使用 git remote add 命令设置 origin 地址。

```
~/exampleapp$ git remote add origin <服务器用户账户>@<服务器IP地址或域名>:exampleapp
```

比如, 服务器用户账户为 `pyrasis`, 服务器 IP 地址为 `192.168.0.40`, `origin` 地址就为 `pyrasis@192.168.0.40:exampleapp`。

使用 `git push` 命令将源代码上传到服务器。

```
~/exampleapp$ git push origin master
```

从 `git push` 命令的输出结果可以看到创建的 Docker 镜像与容器。`git push` 命令完全执行完毕后, 运行 Web 浏览器, 连接服务器的 IP 地址。

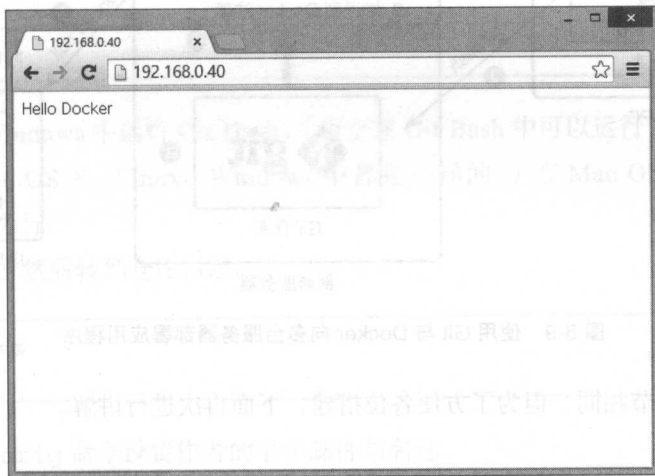


图 8-8 在 Web 浏览器中连接服务器的 Docker 容器

Web 浏览器中显示从 `app.js` 输出的 `Hello Docker` 字符串。接下来, 修改源代码并推送到服务器, 即可创建新的 Docker 容器。

各位根据自身情形修改并使用 `Dockerfile` 与 `post-receive` 文件即可。

8.2 ▶ 向多台服务器部署应用程序

下面学习如何向多台服务器部署应用程序。与在一台服务器中创建 Docker 镜像不同, 由于要将 Docker 镜像传送到多个服务器, 所以必须搭建 Docker 注册服务器。

1. 在开发者 PC 中开发应用程序。
2. 使用 `git push` 命令将源代码上传到部署服务器。
3. 向仓库执行 `git push` 命令时, 部署服务器运行 `git hook`。
4. 在 `git hook` 中创建 Docker 镜像后, 上传到 Docker 注册服务器。

5. 部署服务器使用 SSH 在应用程序服务器中执行 `docker pull` 命令，然后使用 `docker run` 命令创建容器。

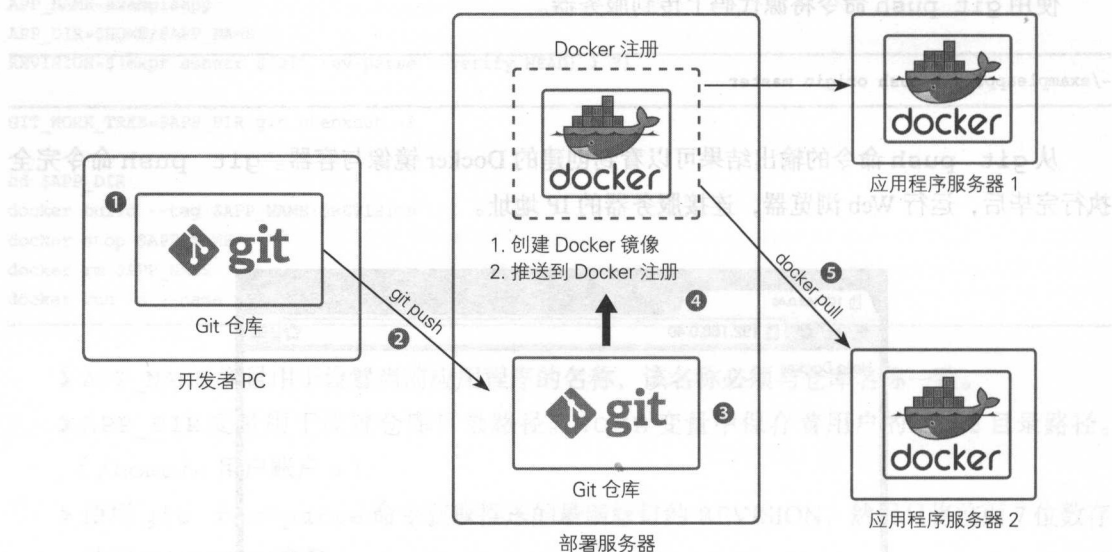


图 8-9 使用 Git 与 Docker 向多台服务器部署应用程序

虽然内容与 8.1 节相同，但为了方便各位搭建，下面再次进行讲解。

8.2.1 在开发者 PC 安装 Git 并创建仓库

若开发 PC 中尚未安装 Git，则请先安装。（<http://git-scm.com>）

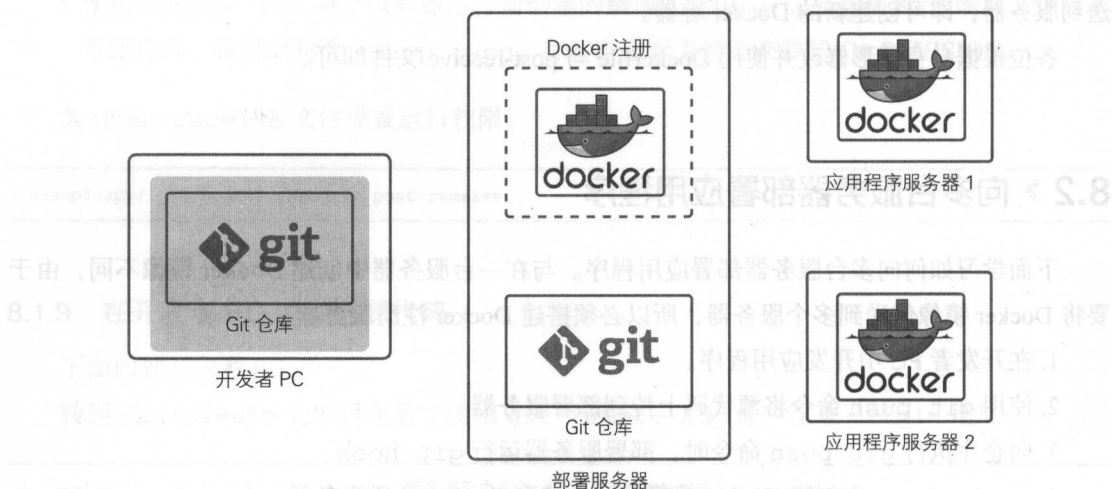


图 8-10 在开发 PC 中安装 Git 并创建代码库

Windows 与 Mac OS X 系统用户从下列网址下载并安装文件即可。安装时没有什么需要特别注意的内容，此处不再赘述。

➤ Windows: <http://msysgit.github.com>

➤ Mac OS X: <http://sourceforge.net/projects/git-osx-installer>

> Ubuntu

```
$ sudo apt-get install git
```

> CentOS

```
$ sudo yum install git
```

接下来，在 Windows 中运行 Git Bash。（由于在 Git Bash 中可以运行 Unix/Linux 命令，所以下述命令在 Mac OS X、Linux、Windows 中都是一样的。）在 Mac OS X 中运行终端，在 Linux 中打开终端运行。

创建 Git 仓库，然后转到仓库目录。

```
~$ git init exampleapp
~$ cd exampleapp
```

使用 `git config` 命令设置作者的电子邮件与名称。

```
~/exampleapp$ git config --global user.email gd@yuldo.com
~/exampleapp$ git config --global user.name "Hong, Gildong"
```

8.2.2 在开发者 PC 中使用 Node.js 编写 Web 服务器

在开发者 PC 中使用 Node.js 编写简单的 Web 服务器。请将如下内容保存为 `app.js`。

➤ `dockerbook/Chapter08/MultipleServerDeployment/exampleapp/app.js`

> `~/exampleapp/app.js`

```
var express = require('express');
var app = express();

app.get(['/', '/index.html'], function (req, res) {
  res.send('Hello Docker');
});

app.listen(80);
```

为了使用 Node.js npm 包，编写如下代码，然后保存为 package.json。

➤ dockerbook/Chapter08/MultipleServerDeployment/exampleapp/package.json

> ~/exampleapp/package.json

```
{
  "name": "exampleapp",
  "description": "Hello Docker",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.x"
  }
}
```

提示 关于 Node.js 的使用方法请参考我编写的文档。

> <http://pyrasis.com/nodejs/nodejs-HOWTO>

使用 git add、git commit 命令将文件提交到开发者 PC 的 exampleapp 仓库。

```
~/exampleapp$ git add app.js package.json
~/exampleapp$ git commit -m "add source"
```

8.2.3 在开发者 PC 中编写 Dockerfile 文件

下面在开发者 PC 中编写 Dockerfile 文件，以在部署服务器中创建并部署 Docker 镜像。请将如下内容保存为 Dockerfile 文件。

➤ dockerbook/Chapter08/MultipleServerDeployment/exampleapp/Dockerfile

> ~/exampleapp/Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y nodejs npm

ADD app.js /var/www/app.js
ADD package.json /var/www/package.json

WORKDIR /var/www
RUN npm install

CMD nodejs app.js
```

- FROM 指令用于设置基于 ubuntu 14.04 创建镜像。
- RUN 指令用于安装 nodejs、npm 包。
- ADD 指令用于将 app.js 与 package.json 复制到镜像的 /var/www 目录。
- WORKDIR 指令用于将执行目录更改为 /var/www。用 RUN 指令运行 npm install 命令，安装 package.json 中设置的 Node.js 模块。
- CMD 设置用于在容器启动时利用 nodejs 运行 app.js。（若 Ubuntu 中以包形式安装 Node.js，则运行文件为 nodejs 而非 node。）

将 Dockerfile 文件也提交到开发者 PC 的 exampleapp 仓库。

```
~/exampleapp$ git add Dockerfile
~/exampleapp$ git commit -m "add Dockerfile"
```

8.2.4 在开发者 PC 中生成 SSH 密钥

在开发者 PC 中生成 SSH 密钥。（在 Windows 系统下要在 Git Bash 中运行如下命令。）

- Enter file in which to save the key：使用默认值。
- Enter passphrase, Enter same passphrase again：点击 Enter 键，不设置密码。

若 id_rsa、id_rsa.pub 文件已经存在于 /home/< 用户账户 >/.ssh 目录，则跳过该部分。

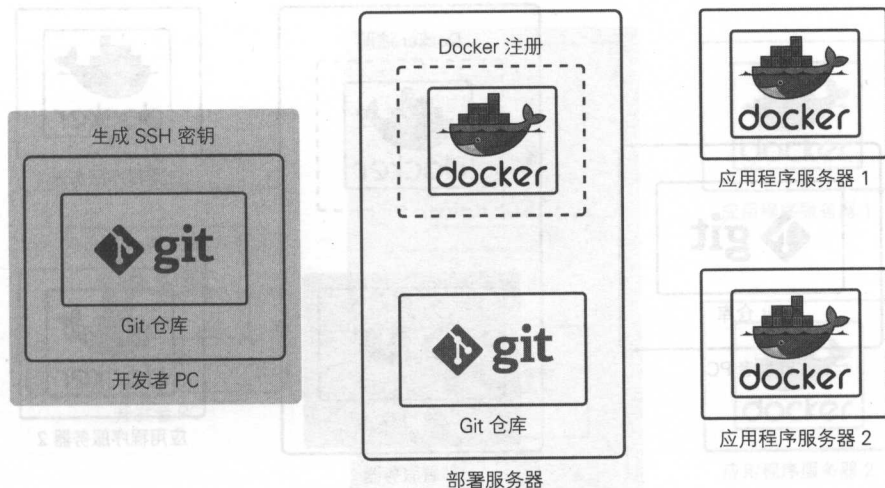


图 8-11 在开发者 PC 中生成 SSH 密钥

```

$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/pyrasis/.ssh/id_rsa):
Created directory '/home/pyrasis/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pyrasis/.ssh/id_rsa.
Your public key has been saved in /home/pyrasis/.ssh/id_rsa.pub.
The key fingerprint is:
64:9e:5a:8a:0b:93:f0:c5:fb:89:41:31:c4:bb:a2:ab pyrasis@ubuntu
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      ..                |
|      ..                |
|      o. o              |
|     ..o + .            |
|    .+. S               |
|   o.+o.o +            |
|  .+.+ o               |
|   o = .                |
| E. o o                 |
+-----+

```

可以看到 /home/<用户账户>/.ssh 目录中生成 id_rsa、id_rsa.pub 文件。

8.2.5 在部署服务器安装 Git 并创建仓库

下面设置部署服务器。由于 Docker 是 Linux 专用的，所以要在 Linux 服务器中进行设置。

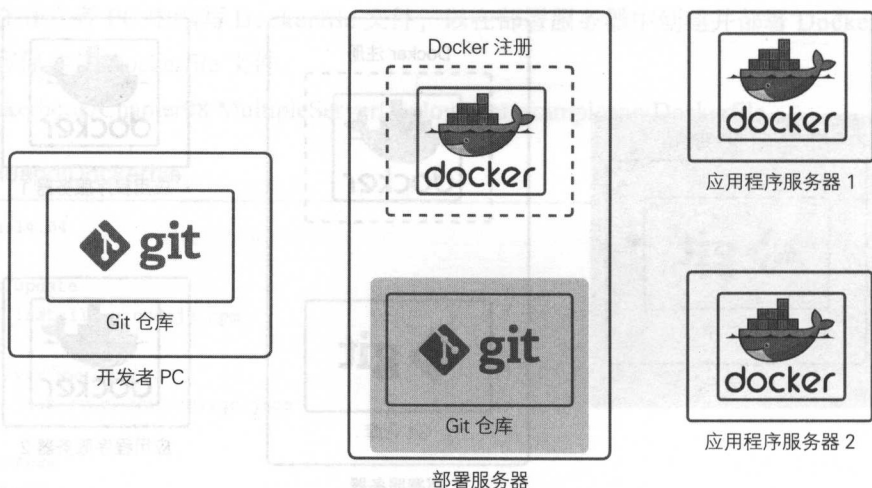


图 8-12 在部署服务器安装 Git 并创建仓库

在部署服务器中也安装 Git。

> Ubuntu

```
$ sudo apt-get install git
```

> CentOS

```
$ sudo yum install git
```

在当前 Linux 用户的 home 目录（/home/< 用户账户 >）下创建 exampleapp 仓库，然后将 receive.denycurrentbranch 设置为 ignore，以便从开发者 PC 接收推送的源代码。

```
~$ git init exampleapp
~$ git config receive.denycurrentbranch ignore
```

8.2.6 在部署服务器中生成 SSH 密钥

若想部署服务器在应用程序服务器中执行 SSH 命令时不必输入密码，需要先生成 SSH 密钥。

在部署服务器中执行 ssh-keygen 命令生成 SSH 密钥。

- Enter file in which to save the key：使用默认值。
- Enter passphrase, Enter same passphrase again：点击 Enter 键，不设置密码。

若 id_rsa、id_rsa.pub 文件已经存在于 /home/< 用户账户 >/.ssh 目录，则跳过该部分。

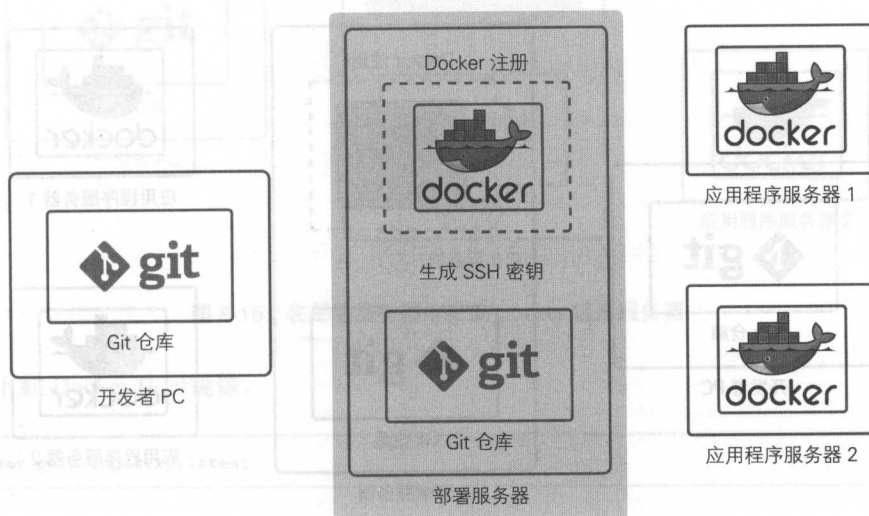


图 8-13 在部署服务器中生成 SSH 密钥

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/pyrasis/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pyrasis/.ssh/id_rsa.
Your public key has been saved in /home/pyrasis/.ssh/id_rsa.pub.
The key fingerprint is:
59:cc:4c:1f:5b:f9:d7:3d:96:36:95:7b:91:09:b8:22 pyrasis@localhost.localdomain
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      . o.o.+      |
|      = o +.=.     |
|      = + .B      |
|      E + . B*     |
|      S . o =      |
|                  |
|                  |
|                  |
+-----+

```

可以看到 /home/< 用户账户 >/.ssh 目录中生成 id_rsa、id_rsa.pub 文件。

8.2.7 在部署服务器中安装 Docker

下面在部署服务器中安装要使用的 Docker。关于在 CentOS 中安装 EPEL 的方法，请参考 2.1.3 节。

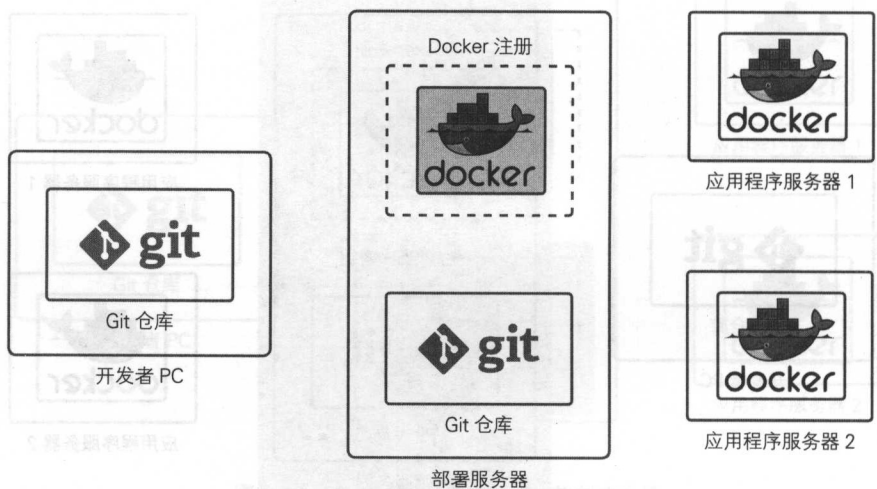


图 8-14 在部署服务器中安装 Docker

> Ubuntu

```
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

> CentOS

```
$ sudo yum install docker-io
$ sudo service docker start
```

使用 `docker` 命令将当前 Linux 用户添加到 `docker` 组，以在不键入 `sudo` 的情形下使用 root 权限。

```
$ sudo usermod -aG docker ${USER}
$ sudo service docker restart
```

8.2.8 在部署服务器中安装 Docker 注册服务器

为了将 Docker 镜像传送到多个应用程序服务器，要安装 Docker 注册（个人仓库）服务器。

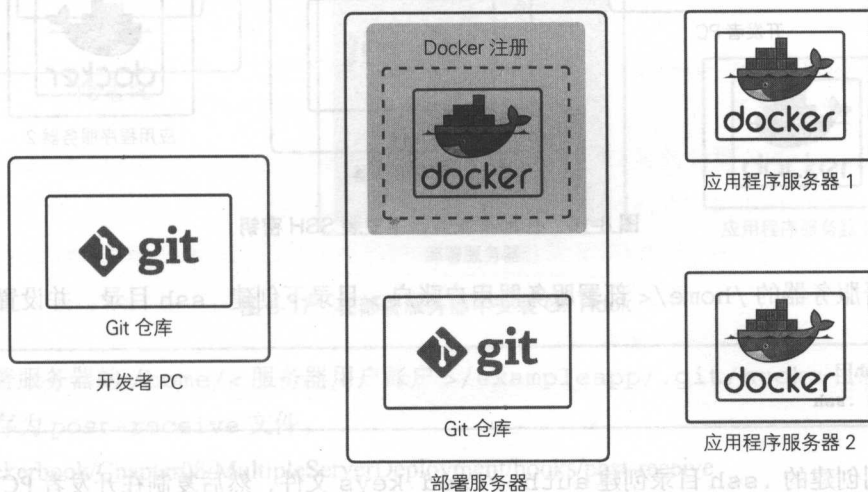


图 8-15 在部署服务器中安装 Docker 注册服务器

首先下载 Docker 注册镜像。

```
$ sudo docker pull registry:latest
```

以容器运行 `registry:latest` 镜像。


```
$ sudo docker run -d -p 5000:5000 --name example-registry \
-v /tmp/registry:/tmp/registry \
registry
```

现在可以在部署服务器的 5000 号端口使用 Docker 注册服务器。

若想将镜像数据存储在 Amazon S3，请参考 6.1.3 节。

8.2.9 在部署服务器中安装 SSH 密钥

下面开始安装前面生成的 SSH 密钥，以从开发者 PC 访问部署服务器时不必输入密码。

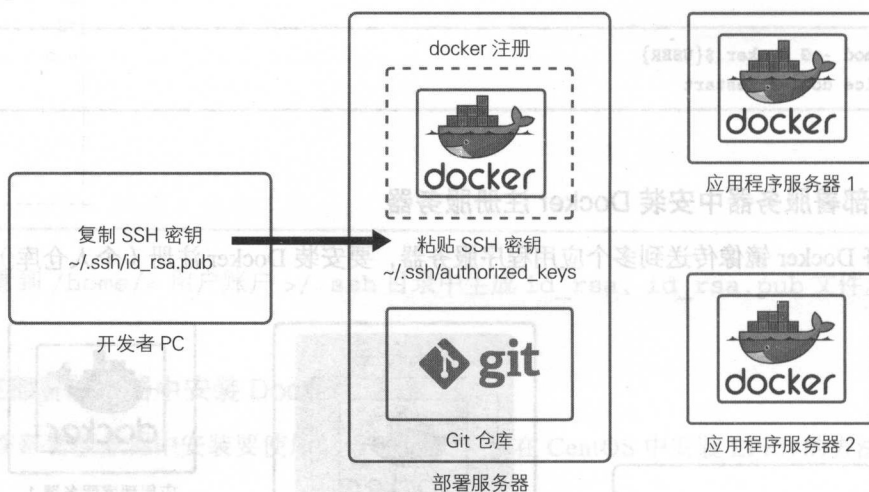


图 8-16 在部署服务器中安装 SSH 密钥

在部署服务器的 `/home/< 部署服务器用户账户 >` 目录下创建 `.ssh` 目录，并设置权限。

```
~$ mkdir .ssh
~$ chmod 700 .ssh
```

在刚刚创建的 `.ssh` 目录创建 `authorized_keys` 文件，然后复制在开发者 PC 中创建的 `id_rsa.pub` 文件的内容，粘贴到 `authorized_keys` 文件。

`> ~/.ssh/authorized_keys`

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCA4nKkAHdB8gU9DT9te9HDNWA8qTY/9YjgxVr493YxqS3vu+Y5/
UyXgRPMEgRZGKWZEDtyssYi/XxQ5Rlk0xXF8NE/T/x8DULVc32e3mtA9vznOMqSHjVLqP0ZZXdhkipEw6s2uKJVTmXQdN+Pz3Dupy
+1a0jww/n3y62goE22f9jr7ZnGtL2haYSZJEMh57RVaywLW3n1Ax53dhKE9ha9xdBC+BRTjkqE8oEq+Vg67H01604kxnRvubiVDIXfm1/
mvXqz+GdgzSta8Uspz59Tth01JocJbAZydl5VEBYV5rziHvRmqwDewV0Mn7hZjAuVYDlqPMbW26/hXsdxycMJ pyrasis@ubuntu
```

以上内容是我的公共密钥。希望各位设置自己的公共密钥 (`id_rsa.pub`)。

为 `authorized_keys` 文件设置权限。

```
$ chmod 600 authorized_keys
```

这样即可在开发者 PC 中随意使用 `git push` 命令而不必输入密码。

8.2.10 在部署服务器中安装 Git Hook

在开发者 PC 中使用 `git push` 命令上传源代码时，要安装 Git Hook 以创建 Docker 镜像。上传到 Docker 注册服务器后，部署到各应用程序服务器。

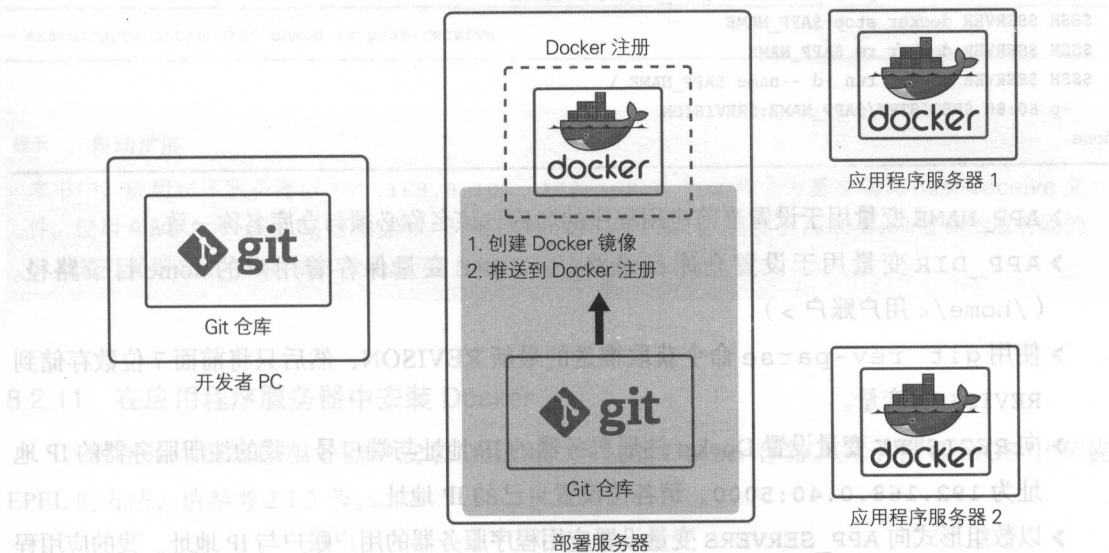


图 8-17 在部署服务器中安装 Git Hook

在部署服务器的 `/home/< 服务器用户账户 >/exampleapp/.git/hooks` 目录中，将以下内容保存为 `post-receive` 文件。

➤ `dockerbook/Chapter08/MultipleServerDeployment/hooks/post-receive`

```
> ~/exampleapp/.git/hooks/post-receive
```

```
#!/bin/bash
```

```
APP_NAME=exampleapp
```

```
APP_DIR=$HOME/$APP_NAME
```

```
REVISION=$(expr substr $(git rev-parse --verify HEAD) 1 7)
```

```
REGISTRY=192.168.0.40:5000
```

```
APP_SERVERS=(
```

```

pyrasis@192.168.0.101"
pyrasis@192.168.0.102"
)

GIT_WORK_TREE=$APP_DIR git checkout -f

cd $APP_DIR
docker build --tag $APP_NAME:$REVISION .
docker tag $APP_NAME:$REVISION $REGISTRY/$APP_NAME:$REVISION
docker push $REGISTRY/$APP_NAME:$REVISION

SSH="ssh -o StrictHostKeyChecking=no"
for SERVER in ${APP_SERVERS[@]}
do
    $SSH $SERVER docker pull $REGISTRY/$APP_NAME:$REVISION
    $SSH $SERVER docker stop $APP_NAME
    $SSH $SERVER docker rm $APP_NAME
    $SSH $SERVER docker run -d --name $APP_NAME \
        -p 80:80 $REGISTRY/$APP_NAME:$REVISION
done

```

- APP_NAME 变量用于设置当前应用程序的名称，该名称必须与仓库名称一致。
- APP_DIR 变量用于设置仓库目录路径。HOME 变量保存着用户的 home 目录路径。（/home/< 用户账户 >）
- 使用 git rev-parse 命令获取推送的最新 REVISION，然后只将前面 7 位数存储到 REVISION 变量。
- 向 REGISTRY 变量设置 Docker 注册服务器的 IP 地址与端口号。我的注册服务器的 IP 地址为 192.168.0.40:5000，请各位设置自己的 IP 地址。
- 以数组形式向 APP_SERVERS 变量设置应用程序服务器的用户账户与 IP 地址。我的应用程序服务器账户与 IP 地址为 pyrasis@192.168.0.101 和 pyrasis@192.168.0.102，请各位设置自己的用户账号与 IP 地址。
- 使用 git checkout -f 命令将推送的源代码保存到仓库目录。必须设置 GIT_WORK_TREE 变量。
- 转到 exampleapp 仓库目录。
- 使用 docker build 命令创建镜像。镜像名称为 exampleapp，标签中设置为刚刚获取的 Git REVISION。
- 为了能够上传到注册服务器，使用 docker tag 命令为刚刚创建的镜像创建标签，格式为 < 注册服务器地址 >/exampleapp:<REVISION>。
- 使用 docker push 命令将镜像上传到注册服务器。
- 向 SSH 变量设置 ssh 命名与选项。只有设置 StrictHostKeyChecking=no 选项才能忽略主机密钥警告，直接运行命令。否则，连接应用程序服务器出现主机密钥警告时，

必须逐个输入 yes。

► 使用循环语句，依据 APP_SERVERS 数组中包含的应用程序服务器的地址，逐一使用 SSH 运行命令。

» 使用 docker pull 命令下载注册服务器中存储的镜像。

» 使用 docker stop、rm 命令停止并删除当前运行的 Docker 容器。

» 使用 docker run 命令以容器运行刚从注册服务器下载的镜像，使用 -p 选项连接 80 号端口，并将其暴露在外。镜像名称为 <注册服务器地址>/exampleapp，标签设置为前面获取的 Git REVISION。

为 post-receive 文件设置运行权限。

```
~/exampleapp/.git/hooks$ chmod +x post-receive
```

提示 自动扩展

本书中，应用程序服务器以 192.168.0.101、192.168.0.102 两个为基准编写 post-receive 文件。使用 AWS Auto Scaling 等服务时，只要使用 AWS API 或 CLI 将属于 Auto Scaling 组的服务器的 IP 地址存储到 APP_SERVERS 变量即可。

8.2.11 在应用程序服务器中安装 Docker

下面在各应用程序服务器中安装 Docker，以创建 Docker 容器。关于在 CentOS 中安装 EPEL 的方法，请参考 2.1.3 节。

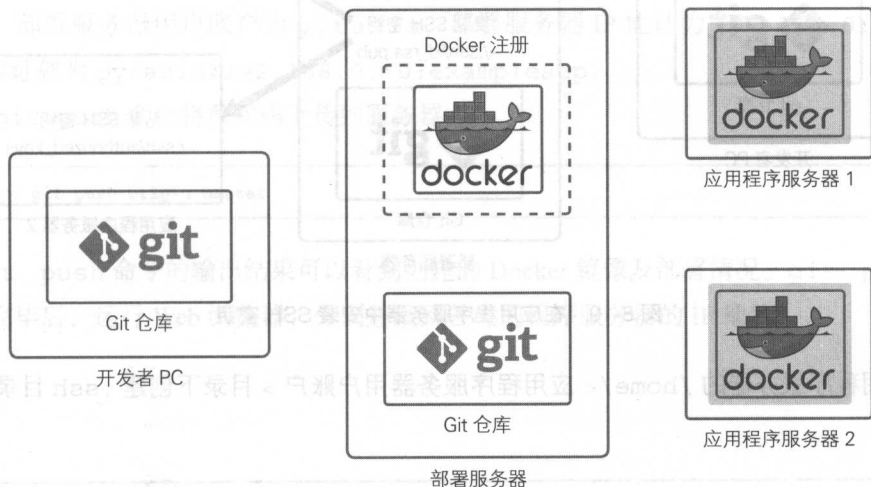


图 8-18 在应用程序服务器中安装 Docker

> Ubuntu

```
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

> CentOS

```
$ sudo yum install docker-io
$ sudo service docker start
```

使用 `docker` 命令将当前 Linux 用户添加到 `docker` 组，即可在不键入 `sudo` 的情形下使用 root 权限。

```
$ sudo usermod -aG docker ${USER}
$ sudo service docker restart
```

8.2.12 在应用程序服务器中安装 SSH 密钥

下面安装前面生成的 SSH 密钥，以从部署服务器访问应用程序服务器时不必输入密码。请注意，必须对每个应用程序服务器都进行设置。

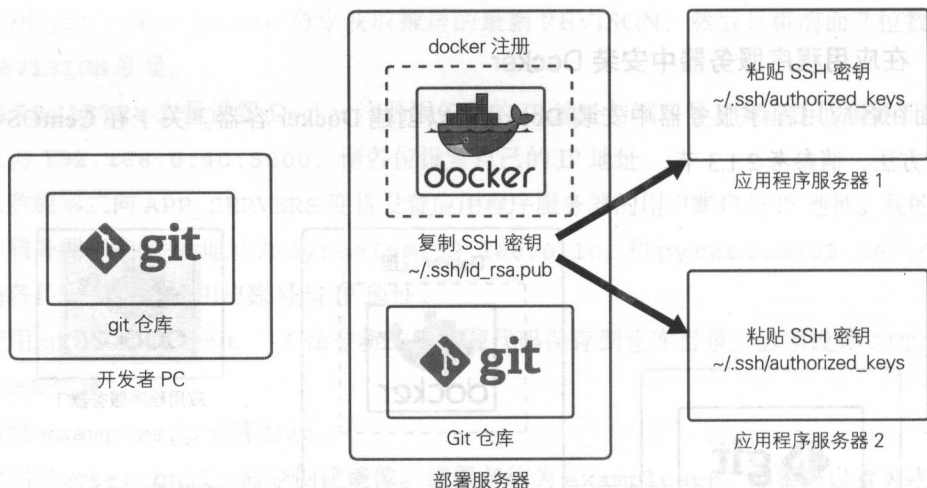


图 8-19 在应用程序服务器中安装 SSH 密钥

在应用程序服务器的 `/home/< 应用程序服务器用户账户 >` 目录下创建 `.ssh` 目录，并设置权限。

```
~$ mkdir .ssh
```

```
~$ chmod 700 .ssh
```

在刚刚创建的 `.ssh` 目录创建 `authorized_keys` 文件，然后将部署服务器中创建 `id_rsa.pub` 文件的内容复制到 `authorized_keys` 文件。

```
> ~/.ssh/authorized_keys
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIWAAQEAtahmHnklgmvyntVT1FEfSn1HhAlLcIpuz/1jzRuVzQj89nGx+hBIM7AHSKp6Azji
cR6/WmvY2hCny2OdmBjkJ9k6Ji8rpOTwlrIkLklkRauonVuZrm0kHCEz1gPpYjaYQG55eQrFF6FJwD4Dyr1cPRw4HFk/RwdSP+q
Kdu2QZn94nOrj851MMYPxzFhdndo/F90pTnXF2YwApNhBh0Njkh/fRBX8qt9qCLvXrGK/ZEs6d8kBJC8HX1zRuweG+op7QkWR7s9
GYlTqOdFYG4MQfLr7K+2RK2qJnjwP3f114t+jJoc2iQ7Hb3g+EbNyDaFwb6Ye2sZJSFXOOv3XDHSXNQ== pyrasis@localhost.
localdomain
```

以上内容是我的公共密钥，请各位设置好自己的公共密钥。（`id_rsa.pub`）

为 `authorized_keys` 文件设置权限。

```
$ chmod 600 authorized_keys
```

这样即可从部署服务器中通过 SSH 使用命令，不必输入密码。

8.2.13 在开发者 PC 中推送源代码

下面回到开发者 PC。

转到 `exampleapp` 仓库目录后，使用 `git remote add` 命令设置 origin 地址。

```
~/exampleapp$ git remote add origin <服务器用户账户>@<服务器IP地址或域名>:exampleapp
```

比如，部署服务器用户账户为 `pyrasis`，部署服务器 IP 地址为 `192.168.0.40`，那么 origin 地址就为 `pyrasis@192.168.0.40:exampleapp`。

使用 `git push` 命令将源代码上传到服务器。

```
~/exampleapp$ git push origin master
```

从 `git push` 命令的输出结果可以看到创建的 Docker 镜像及部署情况。`git push` 命令完全执行完毕后，运行 Web 浏览器，分别连接到各应用程序服务器的 IP 地址。

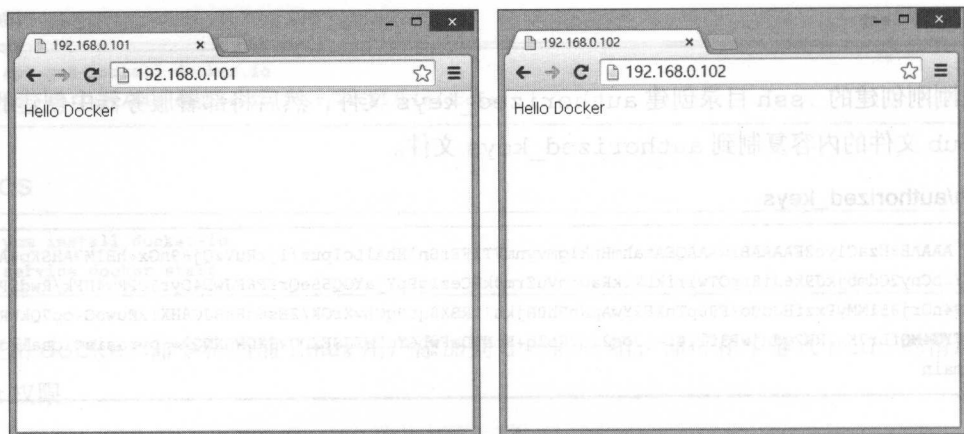


图 8-20 在 Web 浏览器中连接各应用程序服务器的 Docker 容器

Web 浏览器中显示从 `app.js` 输出的 `Hello Docker` 字符串。接下来，修改源代码并推送到服务器，部署新的 Docker 镜像。

各位根据自身情形修改并使用 `Dockerfile` 与 `post-receive` 文件即可。

第9章

DOCKER

Docker 监控

使用 Docker 构建服务时，有必要对服务器的 CPU、内存的使用量进行监控。监控服务器资源的工具有很多，Graphite 就是其中之一。本章将介绍使用 Graphite 工具搭建服务器监控系统的方法。

Graphite 开源工具集合了各种实用工具以监控服务器资源，并将监控结果以图表形式显示。下面介绍如何将 Carbon、Graphite Web、Crafana、Elasticsearch、Diamond 组合到 Graphite，并用 Docker 容器运行。

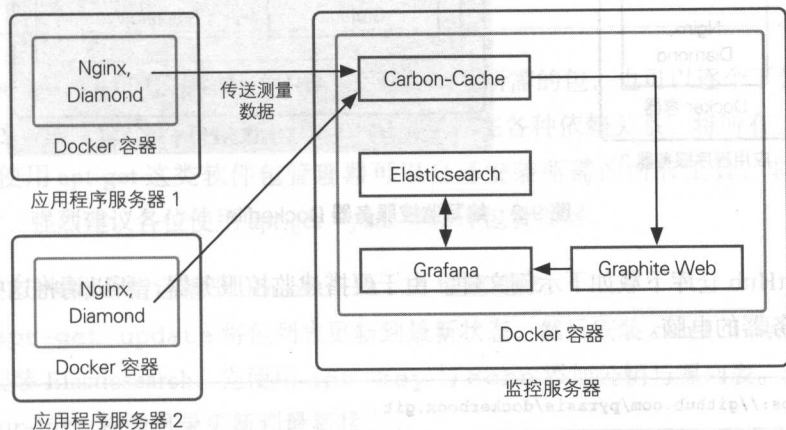


图 9-1 Graphite 监控系统

- ▶ Graphite: 存储基于时间的测量值 (Metric) 数据的仓库，使用 Python 语言编写。
 - » Graphite Web: Web 应用程序，以图表形式显示存储的测量值数据与时间。
 - » Carbon-Cache: 一系列守护进程，从网络收集测量数据并存储到磁盘。
- ▶ Grafana: Graphite 仪表盘，也是 Web 应用程序，其 UI 与画面结构比 Graphite Web 更简

洁。由于要从 Graphite Web 获取数据，故需要与 Graphite Web 配套使用。

» Elasticsearch: 基于 Apache Lucene 的搜索引擎。此处只用于存储 Grafana 仪表盘的图表设置数据。

► Diamond: 收集服务器资源的测量值数据并传送到 Carbon-Cache。收集 CPU、内存、网络使用量与磁盘 I/O 等数据。

请从我的 GitHub 仓库下载示例文件。

► <https://github.com/pyrasis/dockerbook>

9.1 ▸ 编写监控服务器 Dockerfile

首先创建要在监控服务器中使用的 Docker 镜像。

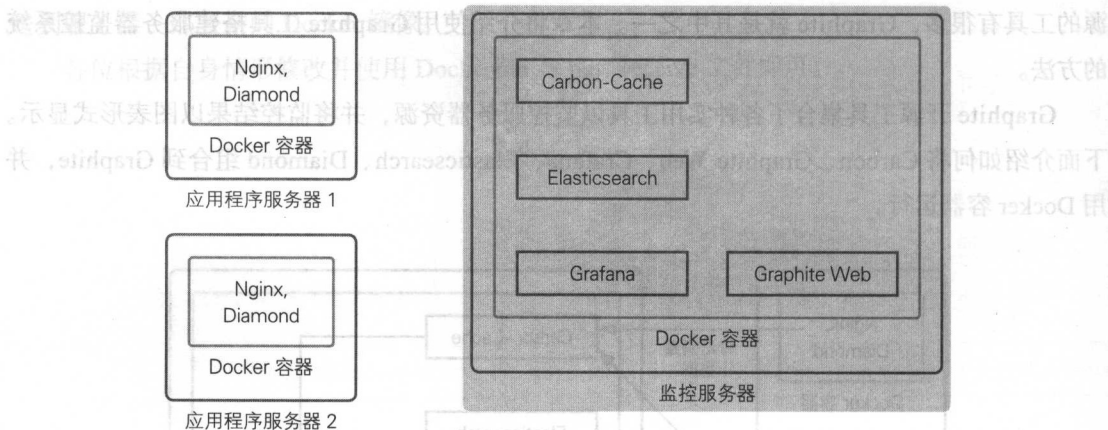


图 9-2 编写监控服务器 Dockerfile

从我的 GitHub 仓库下载如下示例文件。由于要搭建监控服务器，所以请将这些文件下载到要用作监控服务器的电脑。

```
$ git clone https://github.com/pyrasis/dockerbook.git
```

▸ Dockerfile

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get -y install curl
```

```
RUN curl -s http://packages.elasticsearch.org/GPG-KEY-elasticsearch | apt-key add -
```

```

RUN echo "deb http://packages.elasticsearch.org/elasticsearch/1.0/debian stable main" > /etc/apt/
sources.list.d/elasticsearch.list
RUN apt-get update

RUN apt-get install -y graphite-carbon
RUN echo "CARBON_CACHE_ENABLED=true" > /etc/default/graphite-carbon

RUN apt-get install -y graphite-web apache2 apache2-mpm-worker libapache2-mod-wsgi
RUN sudo -u _graphite graphite-manage syncdb --noinput
RUN rm -f /etc/apache2/sites-enabled/000-default.conf
RUN cp /usr/share/graphite-web/apache2-graphite.conf /etc/apache2/sites-enabled/graphite.conf

RUN apt-get install -y elasticsearch openjdk-7-jre-headless
RUN update-rc.d elasticsearch defaults

RUN apt-get install -y nodejs npm
RUN ln -s /usr/bin/nodejs /usr/local/bin/node
RUN curl https://codeload.github.com/grafana/grafana/tar.gz/v1.7.0 | tar -xz
RUN mv grafana-1.7.0 /usr/share/grafana
WORKDIR /usr/share/grafana
RUN npm install
RUN node_modules/grunt-cli/bin/grunt
RUN echo "alias /grafana /usr/share/grafana/src" > /etc/apache2/sites-enabled/grafana.conf
ADD config.js /usr/share/grafana/src/config.js

ADD entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

```

我在 Ubuntu 14.04 中使用 apt-get 命令下载并安装所需的包，也可以逐个下载并安装所需的源代码格式包，或者使用 Python pip 工具。但由于存在各种依赖关系，将所有工具都安装好并非易事。而使用 apt-get 这类软件包管理器可以自动安装所需的所有工具，非常方便。编写 Dockerfile 时，强烈建议各位使用 apt-get、yum 等软件包管理器。

- 使用 FROM 指令指定基于 ubuntu 14.04 创建镜像。
- 使用 apt-get update 将包列表更新到最新状态，然后安装 curl 工具。
- 为了安装 Elasticsearch，先使用 apt-key 与 echo 添加公钥与源列表。然后使用 apt-get update 将包目录更新到最新状态。
- 安装 graphite-carbon 包，将 Carbon-Cache 设置为启用状态。
- 安装 graphite-web 与 Apache Web 服务器包。使用 graphite-manage 创建 SQLite DB 文件，删除默认的 Apache 设置后，复制 Graphite Web 的设置文件。
- 安装 OpenJDK 与 Elasticsearch，使用 update-rc.d 命令设置为以服务启动。
- 安装 nodejs、npm 包，将 /usr/bin/nodejs 可执行文件链接至 /usr/local/bin/node。若不链接至 node，则运行 npm install 的过程中会发生错误。

- ▶ 从 Github 下载 Grafana 源代码，解压缩后，移动到 /usr/share/grafana 目录。
- ▶ 运行 `npm install`，安装 Grafana 所需模块，运行 `grunt` 以创建 CSS。
- ▶ 向镜像添加 `config.js` 文件。
- ▶ 向镜像添加 `entrypoint.sh` 文件。
- ▶ 为镜像的 `/entrypoint.sh` 文件设置运行权限。
- ▶ 使用 `ENTRYPOINT` 命令设置在容器创建时运行 `entrypoint.sh`。

下面是 Grafana 设置文件。

> config.js

```
define(['settings'],
function (Settings) {
    "use strict";

    return new Settings({
        datasources: {
            graphite: {
                type: 'graphite',
                url: "http://192.168.0.40",
            },
            elasticsearch: {
                type: 'elasticsearch',
                url: "http://192.168.0.40:9200",
                index: 'grafana-dash',
                grafanaDB: true,
            }
        },
        search: {
            max_results: 20
        },
        default_route: '/dashboard/file/default.json',
        unsaved_changes_warning: true,
        playlist_timespan: "1m",
        admin: {
            password: ''
        },
        plugins: {
            panels: []
        }
    });
});
```

- ▶ 在 `graphite` 的 `url` 中设置监控服务器域名或 IP 地址。192.168.0.40 是我的监控服务器的 IP 地址，请各位设置为自己的 IP 地址或域名。
- ▶ 在 `elasticsearch` 的 `url` 中设置监控服务器的域名或 IP 地址，端口号设置为 9200。

192.168.0.40 是我的监控服务器的 IP 地址，请各位设置为自己的 IP 地址或域名。

▶ 由于 config.js 文件要用在 Web 浏览器中，故不能设置为 127.0.0.1 这类本地环回地址。

下面是容器启动时要运行的 entrypoint.sh 文件。

> entrypoint.sh

```
#!/bin/bash

service carbon-cache start
service elasticsearch start

apachectl -DFOREGROUND
```

▶ 运行 carbon-cache、elasticsearch、apache2 服务。

▶ 若想以后台模式运行容器，则最后运行的进程必须总在 foreground 处于待机状态。因此，使用 -DFOREGROUND 选项，以 foreground 运行 Apache Web 服务器。

转到 Dockerfile 文件所在目录，使用 docker build 命令创建镜像。

```
~$ cd dockerbook/Chapter09/Graphite
~/dockerbook/Chapter09/Graphite$ sudo docker build --tag graphite .
```

经过短暂等待后，创建 graphite 镜像。依据不同的网络情况，使用 apt-get 或 npm 命令下载文件时可能失败，此时再次运行 docker build 命令。

Graphite 数据是时间序列数据，所以其时间必须准确无误，故使用 ntpdate 命令设置准确时间。若在主机中设置时间，则会自动应用到容器。

```
$ sudo ntpdate time2.kriss.re.kr
```

将 graphite 镜像创建为容器，使用 -d 选项使其以后台模式运行，连接 2003、9200、80 端口后，将其暴露在外。

```
$ sudo docker run -d --name graphite -p 2003:2003 -p 9200:9200 -p 80:80 graphite
```

9.2 ▶ 编写应用程序服务器 Dockerfile

下面创建要在应用程序服务器中使用的 Docker 镜像。

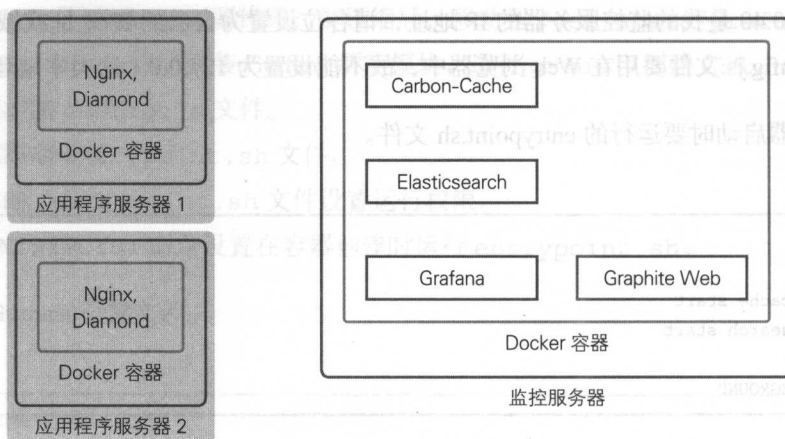


图 9-3 编写应用程序服务器 Dockerfile

从我的 GitHub 仓库的示例文件中下载如下文件。由于要搭建应用程序服务器，所以将这些文件下载到要用作应用程序服务器的各台电脑。

```
$ git clone https://github.com/pyrasis/dockerbook.git
```

> Dockerfile

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y nginx \
    git make pbuilder python-mock python-configobj \
    python-support cdb
```

```
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
```

```
RUN chown -R www-data:www-data /var/lib/nginx
```

```
WORKDIR /tmp
```

```
RUN git clone https://github.com/BrightcoveOS/Diamond.git
```

```
RUN cd Diamond && git checkout v3.4 && make deb
```

```
RUN dpkg -i Diamond/build/diamond_3.4.0_all.deb
```

```
RUN cp /etc/diamond/diamond.conf.example /etc/diamond/diamond.conf
```

```
ADD entrypoint.sh /entrypoint.sh
```

```
RUN chmod +x /entrypoint.sh
```

```
ENTRYPOINT ["/entrypoint.sh"]
```

```
EXPOSE 80
```

```
EXPOSE 443
```

> 使用 FROM 命令指定以 ubuntu:14.04 为基础创建镜像。

- 使用 `apt-get update` 将包目录更新到最新状态，然后安装 `nginx` 包与 `Diamond` 所需的包。
- 设置 `nginx` 以 `foreground` 而非守护进程方式运行，然后将 `/var/lib/nginx` 目录的用户与组设置为 `www-data`。
- 在 `/tmp` 目录下使用 `git` 下载 `Diamond`，将 `v3.4` 标签检出 (`checkout`)。使用 `make deb` 命令创建 `deb` 包文件。使用 `dpkg` 命令安装 `deb` 文件时，文件名包含版本号，所以源代码也选用对应版本。
- 安装使用 `dpkg` 命令构建的 `deb` 文件。
- 将 `Diamond` 示例设置文件复制到默认设置文件。(`diamond.conf`)
- 向镜像添加 `entrypoint.sh` 文件。
- 为镜像的 `/entrypoint.sh` 文件指定运行权限。
- 使用 `ENTRYPOINT` 命令设置在容器创建时运行 `entrypoint.sh`。
- 使用 `EXPOSE` 命令将 `80`、`443` 端口连接到主机。

下面是容器启动时要运行的 `entrypoint.sh` 文件。

> entrypoint.sh

```
#!/bin/bash

sed -i "s/host = graphite/host = $GRAPHITE_HOST/g" /etc/diamond/diamond.conf
diamond

cd /etc/nginx
nginx
```

- 将 `/etc/diamond/diamond.conf` 文件的 `host` 设置为 `$GRAPHITE_HOST` 变量值，可以在 `docker run` 命令中使用 `-e` 选项设置该变量。
- 运行 `diamond`，运行时即以守护进程方式运行。
- 我以 `nginx` 作为示例运行，各位以后运行自己编写的应用程序即可。

转到 `Dockerfile` 所在目录，使用 `docker build` 命令创建镜像。

```
~$ cd dockerbook/Chapter09/Diamond
~/dockerbook/Chapter09/Diamond$ sudo docker build --tag diamond .
```

与监控服务器一样，应用程序服务器也要校准时间。

```
$ sudo ntpdate time2.kriss.re.kr
```

将 diamond 镜像创建为容器。使用 -d 选项以守护进程模式运行，连接到 80 号端口后暴露在外。使用 -e 选项将监控服务器的 IP 地址或域名设置到 GRAPHITE_HOST 变量。192.168.0.40 是我监控服务器的 IP 地址，各位设置为自己的 IP 地址或域名即可。

```
$ sudo docker run -d --name app1 -p 80:80 -e GRAPHITE_HOST=192.168.0.40 diamond
```

9.3 在 Web 浏览器中查看图表

运行 Web 浏览器，连接到监控服务器的 IP 地址或域名。

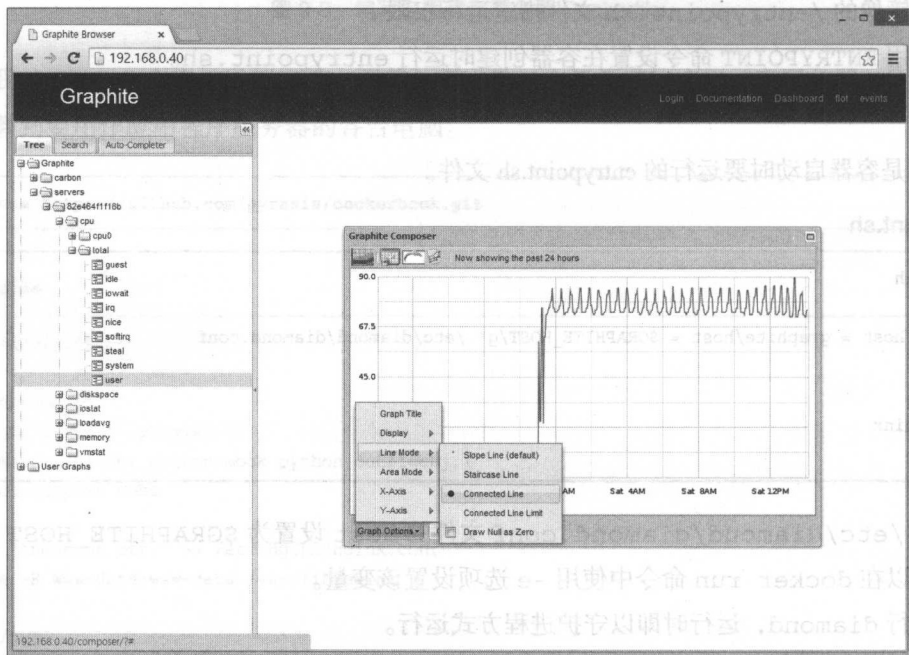


图 9-4 在 Web 浏览器中连接 Graphite Web

在左侧 tree 中，依次进入 Graphite → servers，可以看到刚刚在应用程序服务器中创建的容器的主机名称。单击下层目录，右侧显示相应目录的图表。单击 Graph Options 按钮，在 Line Mode 中单击 Connected Line，显示如图 9-4 所示的连线图表。

由于运行 Diamond 没多久，尚未收集到大量数据，图表可能无法正常显示。图 9-4 是我收集了 14 小时以上的数据而显示的图表。

提示 测试 CPU、内存使用量

要想正常显示图表就需要收集大量数据，为此，如下修改 entrypoin.sh 文件。

> entrypoin.sh

```
#!/bin/bash

sed -i "s/host = graphite/host = $GRAPHITE_HOST/g" /etc/diamond/diamond.conf
diamond

apt-get install -y wget make gcc
wget http://nodejs.org/dist/v0.10.31/node-v0.10.31.tar.gz
tar vxzf node-v0.10.31.tar.gz
cd node-v0.10.31

while :
do
    make
    make test
    make clean
done
```

上述示例下载 Node.js 源代码并编译，重新生成镜像后创建为容器。运行容器几小时即可收集到 CPU、内存使用量的数据。

```
$ sudo docker build --tag compile .
$ sudo docker run -i -t --rm -e GRAPHITE_HOST=192.168.0.40 compile
```

从 Web 浏览器连接到 < 监控服务器的 IP 地址或域名 >/grafana。

10.1 > 在 Amazon EC2 中使用 Docker

Amazon EC2 是 AWS 提供的弹性云服务器。EC2 相当于 Linux 服务器，您可以在 EC2 实例上安装 Docker 并运行容器。

在 AWS 控制台上创建 EC2 实例。在创建实例时，选择 Amazon Linux 2 操作系统，并选择所需的实例类型和配置。创建实例后，按照以下步骤进行设置。

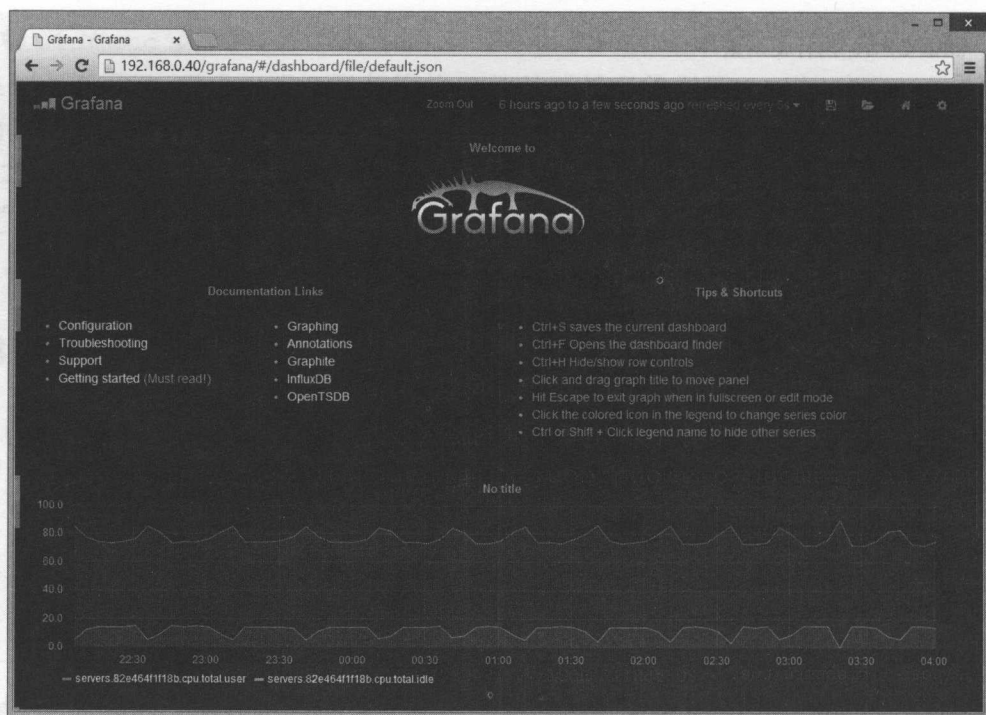


图 9-5 从 Web 浏览器连接到 Grafana

在 Grafana 中也可以查看收集数据的图表。与 Graphite Web 一样，图 9-5 也是我收集了 14 小时数据后得到的图表。

第 10 章

DOCKER

在 Amazon Web Services 中使用 Docker

目前，能够最高效运用 Docker 的场合当数云服务。本章将介绍如何在 Amazon Web Services（以下简称 AWS）的 EC2 与 Elastic Beanstalk 中使用 Docker。

必须拥有 AWS 账号才能使用 EC2 与 Elastic Beanstalk，否则请先创建。AWS 默认为付费服务，但首次加入的用户可以免费使用 1 年 AWS 资源。

提示 加入 AWS 与信用卡验证 1 美元

加入 AWS 时，要验证信用卡是否真实有效，所以会暂时从卡中扣除 1 美元。信用卡通过验证后，会自动返还扣除的 1 美元。

10.2.1 在 AWS 控制台中部署 Docker 应用程序

10.1 在 Amazon EC2 中使用 Docker

Amazon EC2 是 AWS 提供的虚拟服务器。EC2 拥有 User data 功能，在创建实例时运行特定脚本或命令。

在 AWS 控制台创建 EC2 实例。在 3. Configure Instance 中单击 Advanced Details，对 User data 进行设置，如下所示。

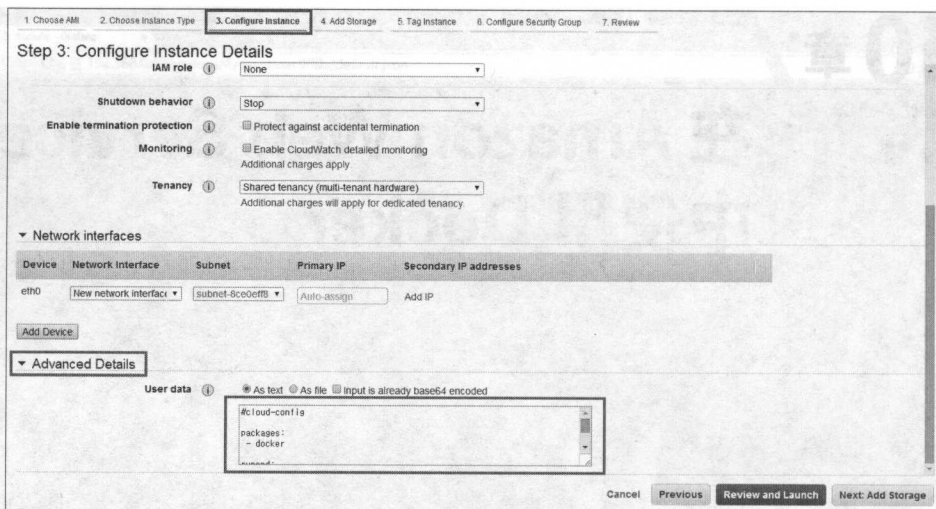


图 10-1 设置 EC2 实例的 User data

在 User data 区块中输入如下内容。

> dockerbook/Chapter10/EC2/AmazonLinux/user-data

> Amazon Linux

```
#cloud-config
```

```
packages:
- docker
```

```
runcmd:
```

```
- [ sh, -c, "usermod -aG docker ec2-user" ]
- service docker start
```

> dockerbook/Chapter10/EC2/Ubuntu/user-data

> Ubuntu

```
#cloud-config
```

```
packages:
- curl
```

```
runcmd:
```

```
- [ sh, -c, "curl https://get.docker.com/ | sh" ]
- [ sh, -c, "usermod -aG docker ubuntu" ]
```

User data 使用 cloud-init 的 Cloud Config 语法，既可以使用 packages 安装包，也可以使用 runcmd 运行命令。

由于 Amazon Linux 还不能使用 `https://get.docker.com` 脚本，所以要使用 `packages` 安装 `docker` 包。Ubuntu 系统安装 `curl` 包后，执行 `https://get.docker.com` 脚本即可。此外，将各 EC2 实例的默认用户添加到 `docker` 组。

注意 对于 Ubuntu 实例，使用 `https://get.docker.com` 脚本安装 Docker 时耗费的时间略长。刚创建实例就使用 SSH 连接时，可能尚未安装 Docker。因此，要暂时等待，直到全部安装完毕。

提示 `cloud-init`、`Cloud Config`

`cloud-init` 是云实例初始化脚本，由创建 Ubuntu Linux 的 Canonical 公司开发。

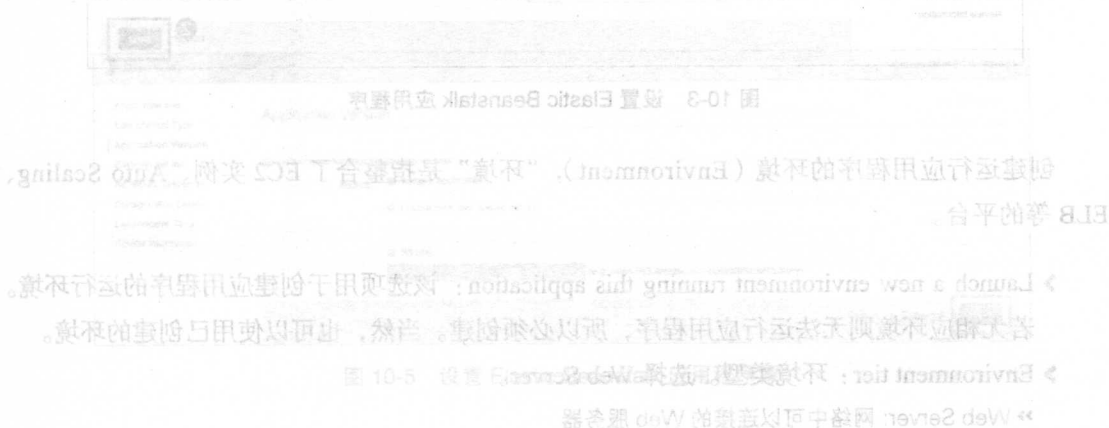
> <https://help.ubuntu.com/community/CloudInit>

10.2 在 AWS Elastic Beanstalk 中使用 Docker

Elastic Beanstalk 提供的平台能够整合 AWS 资源以制作应用程序，它是一种类似于谷歌 App 引擎、VMware Cloud Foundry、Heroku 等的 PaaS（Platform as a Service，平台即服务），既可以运行 Node.js、PHP、Python、Ruby、Java、.NET 应用程序，也可以使用 Docker。

10.2.1 在 AWS 控制台部署 Docker 应用程序

转到 AWS 控制台的 Elastic Beanstalk。没有生成任何 Elastic Beanstalk 应用程序时，显示如图 10-2 所示页面。单击顶部的 **Create New Application**。（在 **Select a Platform** 中选择平台，单击 **Launch Now** 按钮，不需要设置选项，可直接创建。）



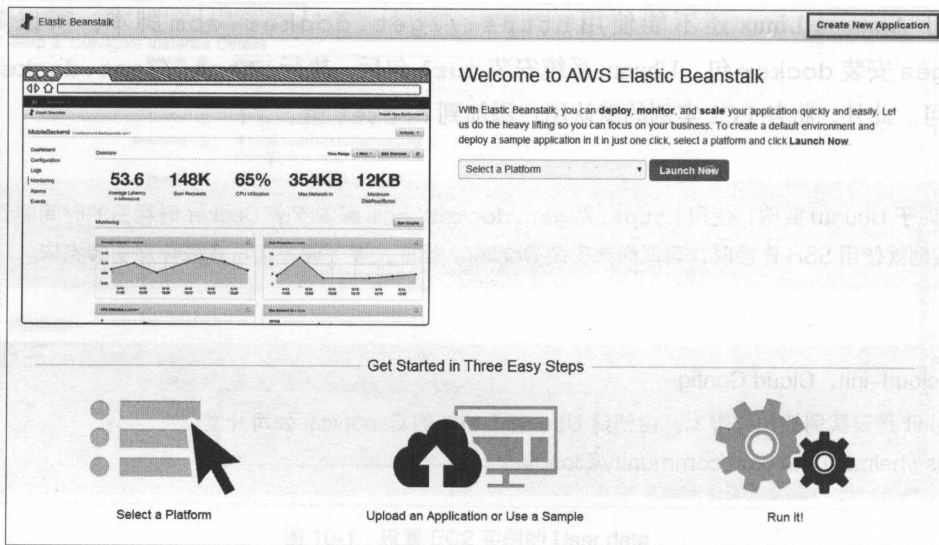


图 10-2 创建 Elastic Beanstalk 应用程序

创建新 Elastic Beanstalk 应用程序。

► **Application Name**：设置应用程序名称，输入 `exampleapp`。

► **Description**：应用程序说明，可不输入。

设置完成后，单击 **Next** 按钮。

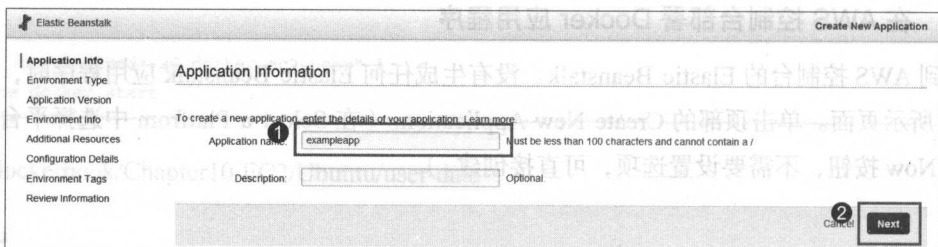


图 10-3 设置 Elastic Beanstalk 应用程序

创建运行应用程序的环境（Environment），“环境”是指整合了 EC2 实例、Auto Scaling、ELB 等的平台。

► **Launch a new environment running this application**：该选项用于创建应用程序的运行环境。若无相应环境则无法运行应用程序，所以必须创建。当然，也可以使用已创建的环境。

► **Environment tier**：环境类型。选择 **Web Server**。

» **Web Server**：网络中可以连接的 Web 服务器

- » Worker: 后台作业环境, 该环境不连接到网络。Worker 与 Web Server 通过 SQS (Simple Queue Service) 交换数据。
- » Predefined configuration: 开发语言或平台。选择 Docker。
- » Environment Type: 环境构成方式。使用默认值。
 - » Load Balancing, autoscaling: 使用负载平衡与自动扩展。
 - » Single Instance: 只使用 1 个 EC2 实例。

设置完成后, 单击 Next 按钮。

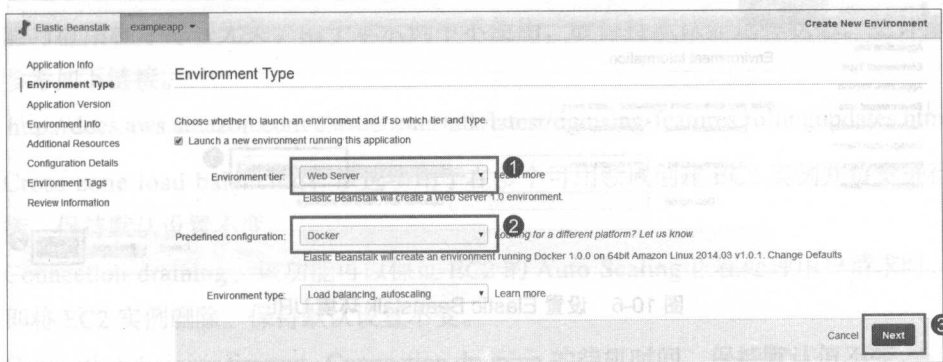


图 10-4 设置 Elastic Beanstalk 环境

上传要运行的应用程序源代码, 或者使用示例应用程序。选择使用默认的示例应用程序。

- » Sample application: Elastic Beanstalk 提供的示例源代码。
- » Upload your own: 上传用户的源代码。
- » S3 URL: 使用 S3 存储区中存储的源代码。

单击 Next 按钮。

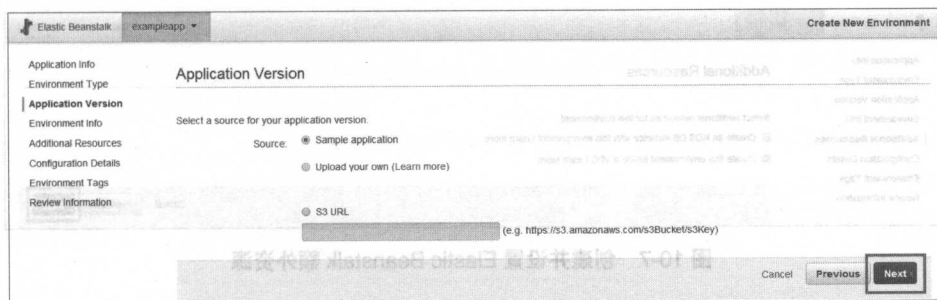


图 10-5 设置 Elastic Beanstalk 应用程序源

设置 Elastic Beanstalk 环境的 URL。从 Web 浏览器访问该 URL，以使用应用程序。

- Environment name：环境名称，使用默认值。
- Environment URL：环境的 URL。URL 必须唯一，所以单击 Check availability 按钮检查是否重复。若重复，请输入其他 URL。
- Description：环境说明，可不输入。

设置完成后，单击 Next 按钮。

图 10-6 设置 Elastic Beanstalk 环境 URL

创建额外资源并进行设置。

- Create an RDS DB Instance with this environment：该选项用于创建 RDS DB 实例。由于本示例中不需要创建 DB 实例，故取消默认选择。
- Create this environment inside a VPC：该选项用于在其他隔离的 VPC 中创建环境。禁止从外部连接而只从内部连接时使用，比如，使用 VPN 连接 VPC 构建公司内部网络时，可以灵活应用。本示例中，由于需要从互联网进行连接，故取消默认选择。

设置完成后，单击 Next 按钮。

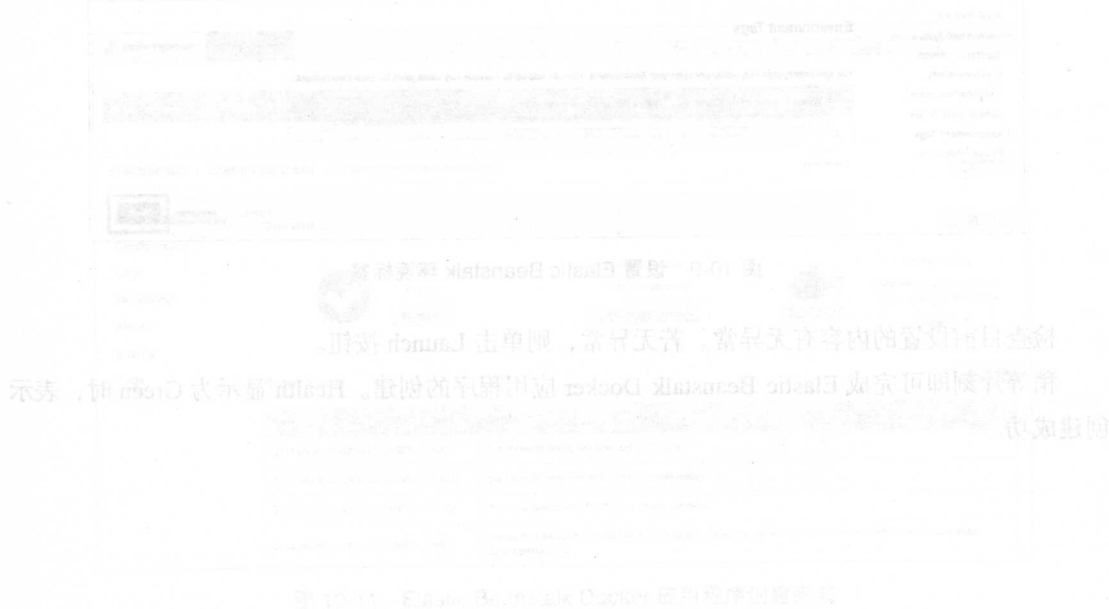
图 10-7 创建并设置 Elastic Beanstalk 额外资源

Elastic Beanstalk 环境详细设置。

- Instance Type：EC2 实例类型，使用默认值。

- ▶ **EC2 key pair**：连接 EC2 实例时要使用的密钥对。
- ▶ **Email address**：环境中主要内容发生变化时，将其发送到指定的电子邮箱。也可以不输入电子邮箱。
- ▶ **Application health check URL**：该 URL 用于检测 ELB 中 Web 服务器（Node.js、Apache Web 服务器、Nginx）是否正常运行。若不输入，则使用 / 路径。保持默认清空状态。
- ▶ **Enable rolling updates**：选择该项将分段使用更新。更改 EC2 实例类型或更换 EC2 实例时，停止 EC2 实例，暂时中断服务。为了不中断服务，需要在部分 EC2 实例处于运行状态时进行局部更新。在完成更新的 EC2 实例中处理通信量，也更新其他 EC2 实例。该功能与应用程序部署无关。由于本示例中不使用，故保持默认非选择状态。更详细内容请参考如下链接。
<http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.rollingupdates.html>
- ▶ **Cross zone load balancing**：该选项用于在多个可用领域创建 EC2 实例并负责进行负载均衡。保持默认设置不变。
- ▶ **Connection draining**：该功能可以保证 EC2 的 Auto Scaling 正在处理用户请求时，不会立即将 EC2 实例删除。保持默认设置不变。
- ▶ **Connection draining timeout**：Connection draining 的待机时间。保持默认值不变。
- ▶ **Instance profile**：EC2 实例中要使用的 IAM。既可以选择已创建的 IAM，也可以重新创建。选择默认的 Create Default Profile。

设置完成后，单击 Next 按钮。



Configuration Details

Modify the following settings or click Continue to accept the default configuration. Learn more.

Instance type: t1.micro
Determines the processing power of the servers in your environment.

EC2 key pair: Select a key pair Refresh
Optional: Enables remote login to your instances.

Email address: Optional: Get notified about any major changes to your environment.

Application health check URL: Enter the relative URL that ELB continually monitors to ensure your application is available.

Enable rolling updates: Lets you control how changes to the environment's instances are propagated. Learn more.

Cross zone load balancing: Enables load balancing across multiple Availability Zones. Learn more.

Connection draining: Enables the load balancer to maintain connections to an Amazon EC2 instance to complete in-progress requests while also stopping new requests. Learn more.

Connection draining timeout: 20 seconds
Maximum time that the load balancer maintains connections to an Amazon EC2 instance before forcibly closing connections.

Instance profile: Create Default Profile Refresh
Grants your environment specific permissions under your AWS account. Learn more.

Cancel Previous **Next**

图 10-8 Elastic Beanstalk 环境详细设置

设置 Elastic Beanstalk 环境标签，最多可以创建 7 个。单击 Next 按钮。

Environment Tags

You can specify tags (key-value pairs) for your Environment. You can add up to 7 unique key-value pairs for each Environment.

Key (128 characters maximum)	Value (256 characters maximum)
1.	

7 remaining

Cancel Previous **Next**

图 10-9 设置 Elastic Beanstalk 环境标签

检查目前设置的内容有无异常。若无异常，则单击 Launch 按钮。

稍等片刻即可完成 Elastic Beanstalk Docker 应用程序的创建。Health 显示为 Green 时，表示创建成功。

The screenshot shows the 'Review' step of creating an Elastic Beanstalk environment. The left sidebar lists navigation options: Application Info, Environment Type, Application Version, Environment Info, Additional Resources, Configuration Details, Environment Tags, and Review Information (selected). The main content area is divided into sections: Application Info (Application name: exampleapp), Environment Type (Tier: Web Server 1.0, Container type: 64bit Amazon Linux 2014.03 v1.0.1 running Docker 1.0.0, Environment type: Load balancing, autoscaling), Application Version (Application source: Sample application), Environment Info (Environment name: exampleapp-env, Environment URL: http://exampleapp-env.elasticbeanstalk.com), Configuration Details (Instance type: t1.micro, Instance profile: aws-elasticbeanstalk-ec2-role, Key pair, Email address, Application health check URL), and Environment Tags (No settings provided). At the bottom right are 'Cancel', 'Previous', and 'Launch' buttons.

图 10-10 检查 Elastic Beanstalk 应用程序及环境设置

The screenshot shows the 'Overview' page of the Elastic Beanstalk environment 'exampleapp'. An info box at the top states: 'Elastic Beanstalk is now creating your environment. When it has finished it will be running Sample Application.' The left sidebar lists navigation options: Dashboard, Configuration, Logs, Monitoring, Alarms, Events, and Tags. The main content area includes:

- Health:** A green checkmark icon and 'Green' status with a 'Monitor' button.
- Running Version:** 'Sample Application' with an 'Upload and Deploy' button.
- Configuration:** '64bit Amazon Linux 2014.03 v1.0.1 running Docker 1.0.0' with an 'Edit' button.
- Recent Events:** A table showing the last four events.

Time	Type	Details
2014-08-25 23:49:00 UTC+0900	INFO	Environment health has been set to GREEN
2014-08-25 23:48:58 UTC+0900	INFO	Successfully launched environment: exampleapp-env
2014-08-25 23:48:32 UTC+0900	INFO	Adding instance 'i-61086778' to your environment.
2014-08-25 23:47:55 UTC+0900	INFO	Added EC2 Instance 'i-61086778' to Auto Scaling Group 'aws-ec2-elasticbeanstalk-AWSEBAutoScalingGroup-1XQ489WIML9UG'

图 10-11 Elastic Beanstalk Docker 应用程序创建完成

下面使用 Node.js 编写简单的 Web 服务器。将如下内容保存为 `app.js`。

➤ `dockerbook/Chapter10/ElasticBeanstalk/app.js`

> `app.js`

```
var express = require('express');
var app = express();

app.get(['/', '/index.html'], function (req, res) {
  res.send('Hello Docker');
});

app.listen(80);
```

为了使用 Node.js npm 包，编写如下代码并保存为 `package.json`。

➤ `dockerbook/Chapter10/ElasticBeanstalk/package.json`

> `package.json`

```
{
  "name": "exampleapp",
  "description": "Hello Docker",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.x"
  }
}
```

将如下内容保存为 `Dockerfile` 文件。

➤ `dockerbook/Chapter10/ElasticBeanstalk/Dockerfile`

> `Dockerfile`

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y nodejs npm

ADD app.js /var/www/app.js
ADD package.json /var/www/package.json

WORKDIR /var/www
RUN npm install

CMD nodejs app.js
EXPOSE 80
```

注意 Dockerfile 必须以 UNIX 换行 (LF) 保存。若以 Windows 换行 (CR/LF) 保存, 则在 Elastic Beanstalk 中部署应用程序时会发生错误。

在 Windows 中使用 AcroEdit (<http://www.acrosoft.pe.kr>) 时, 可以将文件以 UNIX 换行进行保存。使用 Vim 时运行如下命令, 则可以将文件以 UNIX 换行进行保存。

```
:set ff=unix
```

将 app.js、package.json、Dockerfile 文件压缩为 ZIP 文件, 我将它们压缩为 exampleapp.zip。在 Elastic Beanstalk 环境页面中, 单击 Upload and Deploy 按钮。

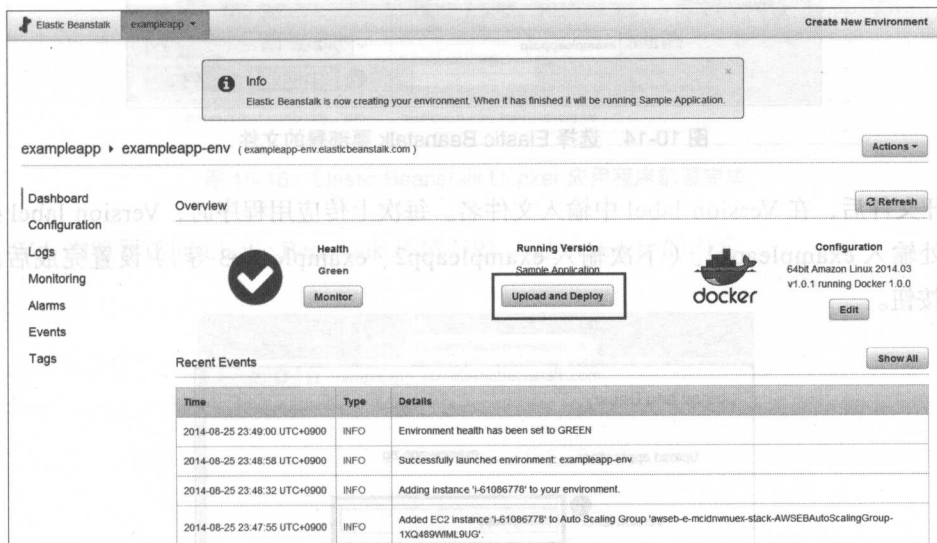


图 10-12 部署 Elastic Beanstalk Docker 应用程序

弹出 Upload and Deploy 窗口, 单击“选择文件”按钮。

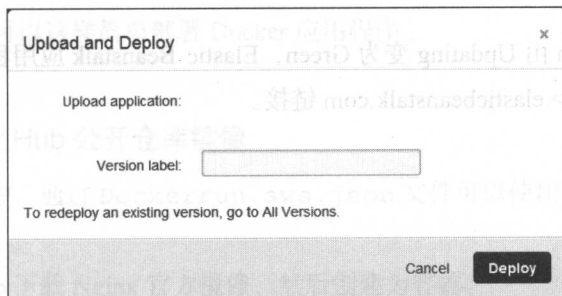


图 10-13 部署 Elastic Beanstalk

在“打开文件”窗口中选择刚刚压缩的 exampleapp.zip 文件, 单击“打开”按钮。

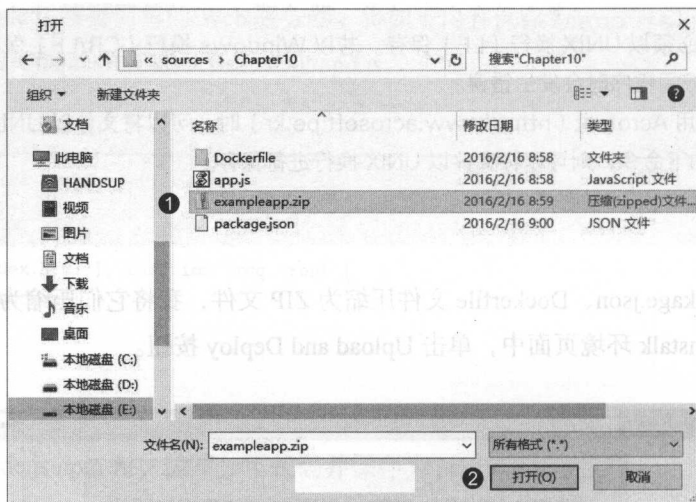


图 10-14 选择 Elastic Beanstalk 要部署的文件

打开文件后，在 Version label 中输入文件名。每次上传应用程序时，Version label 必须不同。此处输入 exampleapp1。（下次输入 exampleapp2、exampleapp3 等。）设置完成后，单击 Deploy 按钮。

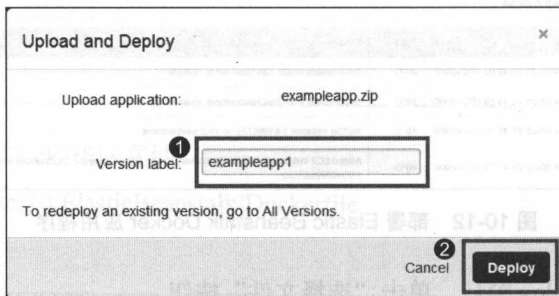


图 10-15 Elastic Beanstalk 部署

短暂等待后，Health 由 Updating 变为 Green，Elastic Beanstalk 应用程序部署完成。接下来，单击上方的 <环境名称>.elasticbeanstalk.com 链接。

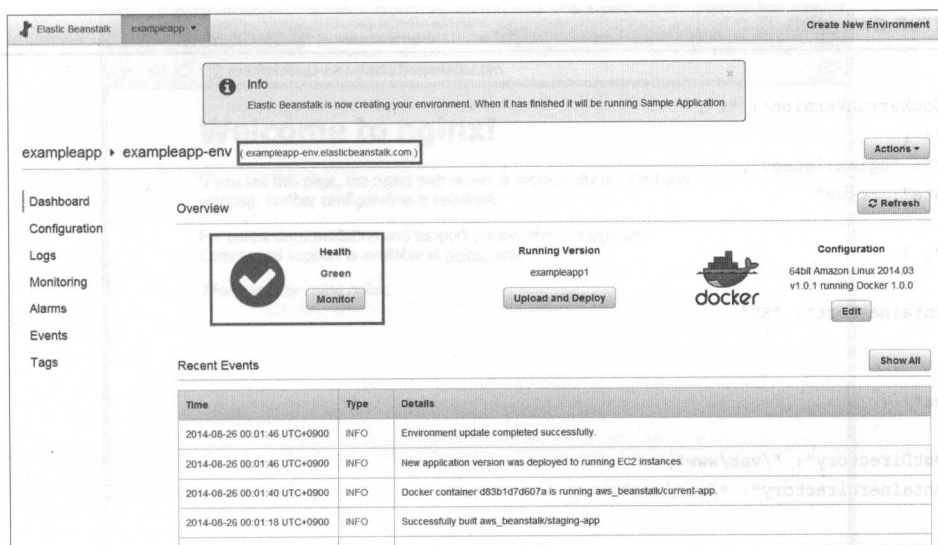


图 10-16 Elastic Beanstalk Docker 应用程序部署完毕

从 Web 浏览器访问 Elastic Beanstalk 环境 URL，显示 app.js 的内容。

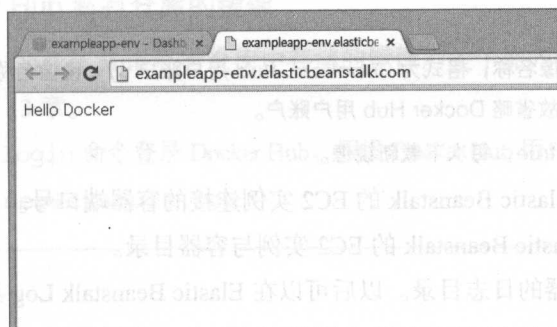


图 10-17 连接 Elastic Beanstalk Docker 应用程序

在 AWS 控制台中可以这样简单部署 Docker 应用程序。

10.2.2 使用 Docker Hub 公开仓库镜像

Elastic Beanstalk 中，通过 `Dockerrun.aws.json` 文件可以使用 Docker Hub 公开仓库中存储的镜像。

下面从 Docker Hub 下载 Nginx 官方镜像，然后创建为容器。

➤ `dockerbook/Chapter10/ElasticBeanstalk/Dockerrun.aws.json`

> Dockerrun.aws.json

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "nginx:latest",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "80"
    }
  ],
  "Volumes": [
    {
      "HostDirectory": "/var/www",
      "ContainerDirectory": "/var/www"
    }
  ],
  "Logging": "/var/log/nginx"
}
```

► **AWSEBDockerrun Version** : Dockerrun 版本号，设置为 1。

► **Image** : 设置镜像。

‣ Name: Docker 镜像名称，格式为 <Docker Hub 用户账户>/<镜像名称>:<标签>。由于要使用 Nginx 官方镜像，故省略 Docker Hub 用户账户。

‣ 将 Update 设置为 true，每次下载新镜像。

► **Ports** : 设置要与 Elastic Beanstalk 的 EC2 实例连接的容器端口号。

► **Volumes** : 连接 Elastic Beanstalk 的 EC2 实例与容器目录。

► **Logging** : 设置容器的日志目录。以后可以在 Elastic Beanstalk Log 菜单中查看日志。

在 Elastic Beanstalk 环境页面单击 Upload and Deploy 按钮，上传 Dockerrun.aws.json 文件后即可创建 Docker 容器。

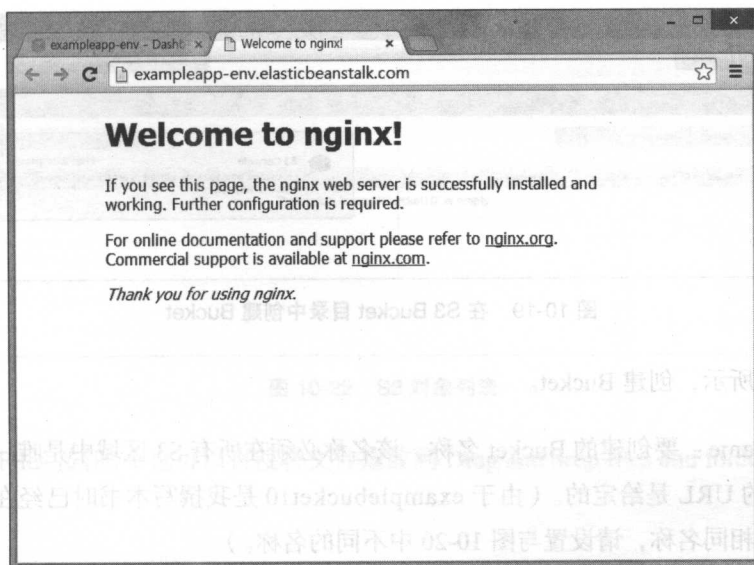


图 10-18 连接 Elastic Beanstalk 中使用公开仓库镜像创建的容器

10.2.3 使用 Docker Hub 私有仓库的镜像

下面学习如何使用 Docker Hub 私有仓库中存储的镜像。关于加入 Docker Hub 与创建私有仓库的方法，请参考第 13 章。

首先使用 `docker login` 命令登录 Docker Hub。假设 Docker Hub 用户账户为 `exampleuser`，电子邮箱为 `exampleuser@example.com`。

```
~$ sudo docker login
Username: exampleuser
Password:
Email: exampleuser@example.com
Login Succeeded
```

登录后，在当前 Linux 用户的 `/home/< 用户账户 >` 创建 `.dockercfg` 文件。

```
> ~/.dockercfg
```

```
{ "https://index.docker.io/v1/": { "auth": "ZKhabXBstTXWzZXL4dnRn0TU2M12=", "email": "exampleuser@example.com" } }
```

查看 `.dockercfg` 文件，可以看到其中保存的验证密钥与电子邮箱。

接下来，在 AWS 中创建 S3 Bucket。在 S3 页面中单击上方的 `Create Bucket` 按钮。

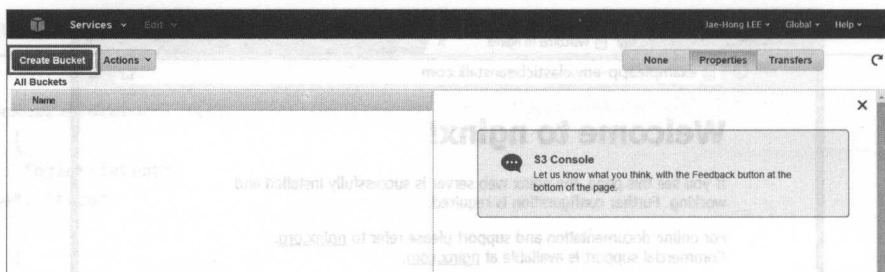


图 10-19 在 S3 Bucket 目录中创建 Bucket

如图 10-20 所示，创建 Bucket。

- **Bucket Name**: 要创建的 Bucket 名称。该名称必须在所有 S3 区域中是唯一的，因为可以访问 S3 的 URL 是给定的。（由于 examplebucket10 是我撰写本书时已经在使用的，所以不能使用相同名称，请设置与图 10-20 中不同的名称。）
- **Region**: Bucket 是依据不同区域创建的。选择 Tokyo。

设置完成后，单击 Create 按钮。若已经在使用设置的 Bucket，请尝试输入其他名称。

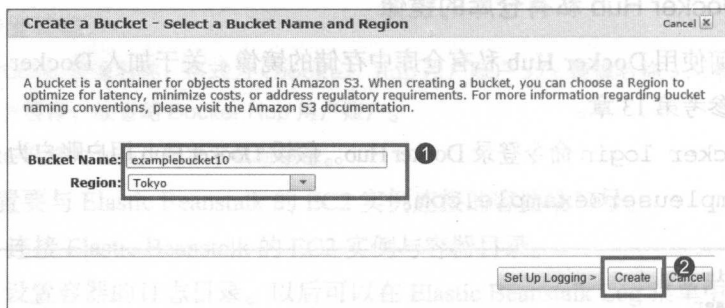


图 10-20 创建 S3 Bucket

单击创建的 Bucket。

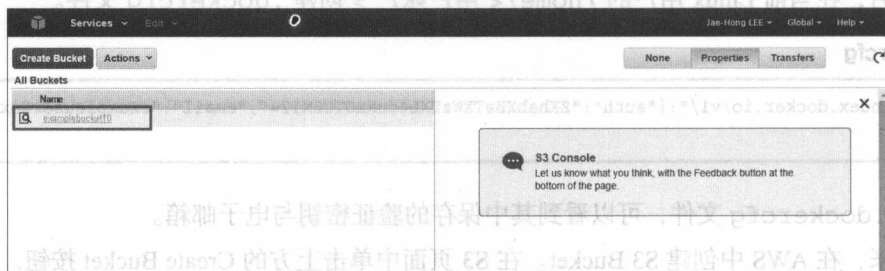


图 10-21 S3 Bucket 创建完成

单击上方的 Upload 按钮。

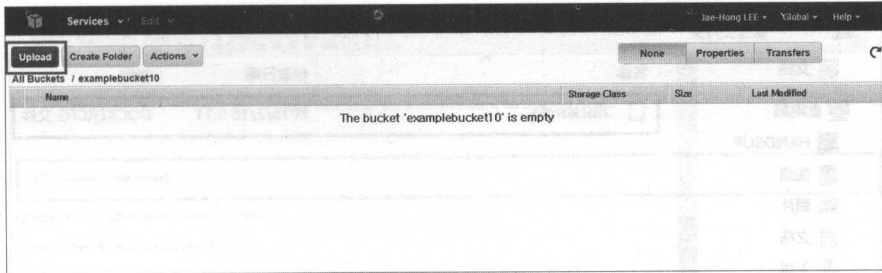


图 10-22 S3 对象列表

单击 Add Files 按钮。(也可以直接将文件拖放到 Drag and drop files and folders to upload here 区域。)

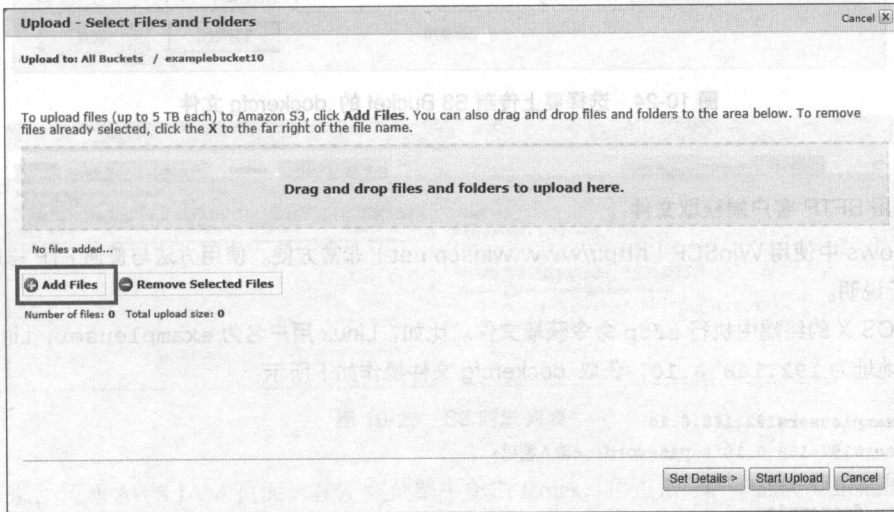


图 10-23 向 S3 Bucket 上传文件

弹出打开文件窗口。我使用 SFTP 客户端将 .dockercfg 文件从安装有 Docker 的 Linux 服务器传送到 Windows 电脑。当然也可以不传送文件，直接将文件从 Linux 服务器上传到 S3 Bucket。

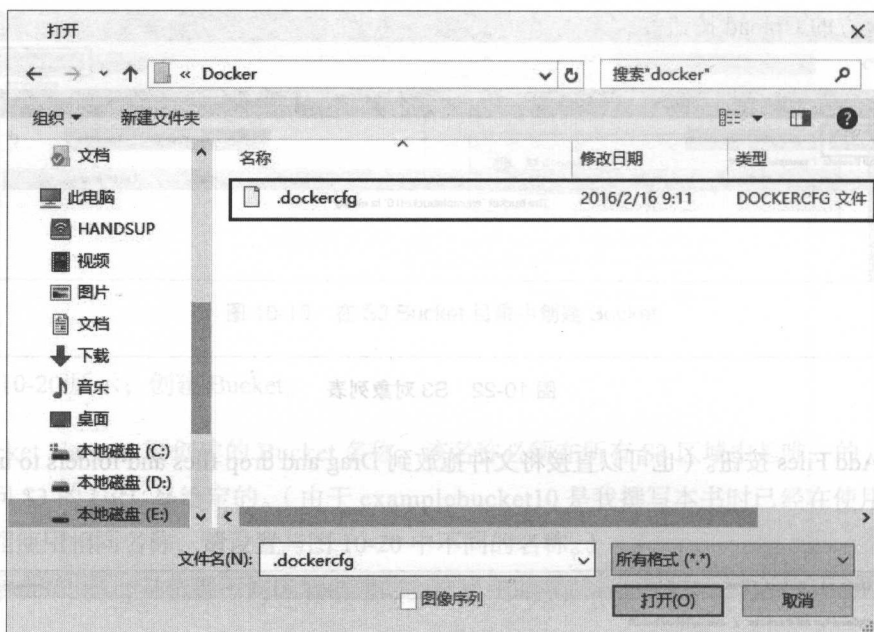


图 10-24 选择要上传到 S3 Bucket 的 .dockercfg 文件

提示 使用 SFTP 客户端获取文件

在 Windows 中使用 WinSCP (<http://www.winscp.net>) 非常方便。使用方法与普通 FTP 程序一样，无需另行说明。

在 Mac OS X 的终端中执行 `sftp` 命令获取文件。比如，Linux 用户名为 `exampleuser`，Linux 服务器的 IP 地址为 `192.168.0.10`，获取 `.dockercfg` 文件操作如下所示：

```
$ sftp exampleuser@192.168.0.10
exampleuser@192.168.0.10's password: <输入密码>
Connected to 192.168.0.10.
sftp> get .dockercfg
Fetching /home/exampleuser/.dockercfg to .dockercfg 100% 99 0.1KB/s 00:00
/home/exampleuser/.dockdrfcg
```

若输入 `sudo docker login` 命令使用 `root` 权限执行命令，则 `.dockercfg` 文件的所有者与组也将是 `root`，这样就无法以普通用户身份复制文件。因此，在安装 Docker 的 Linux 服务器中输入如下命令，将所有者与组更改为当前用户。

```
$ sudo chown $USER.$USER .dockercfg
```

将 `.dockercfg` 文件添加到待上传目录。单击 **Start Upload** 按钮。

图 10-21 S3 Bucket 创建完成

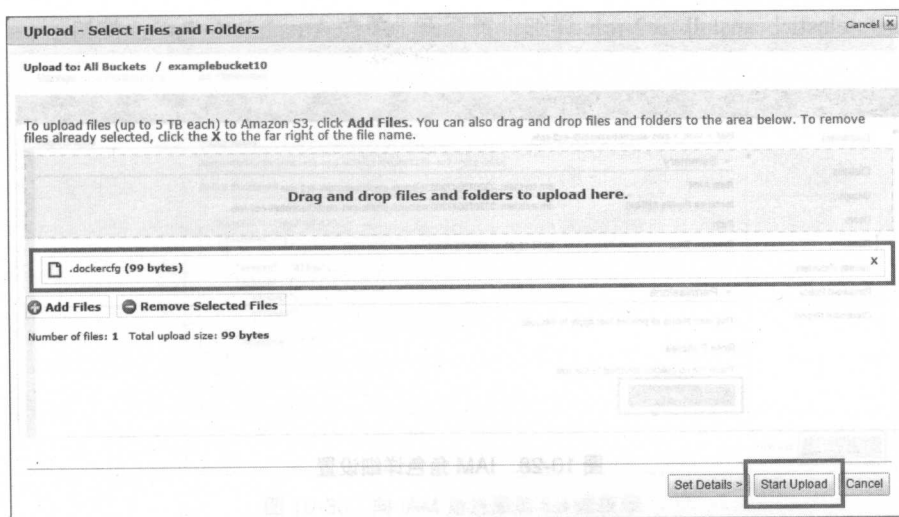


图 10-25 待上传至 S3 Bucket 的文件目录

文件上传到 S3 Bucket。

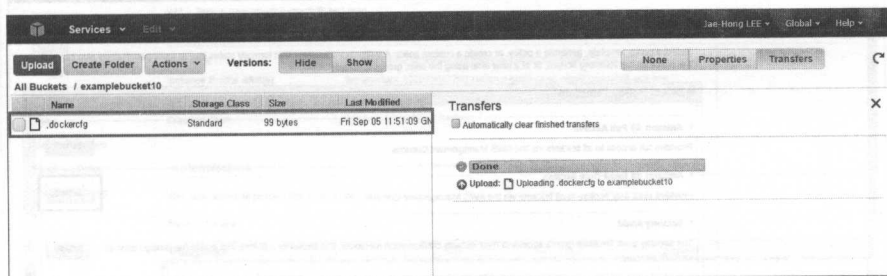


图 10-26 S3 对象列表

接下来，转到 AWS IAM 页面。在左侧菜单中单击 Roles，再点击 aws-elasticbeanstalk-ec2-role。若前面已创建 Elastic Beanstalk 环境与应用程序，则会出现 aws-elasticbeanstalk-ec2-role 选项。

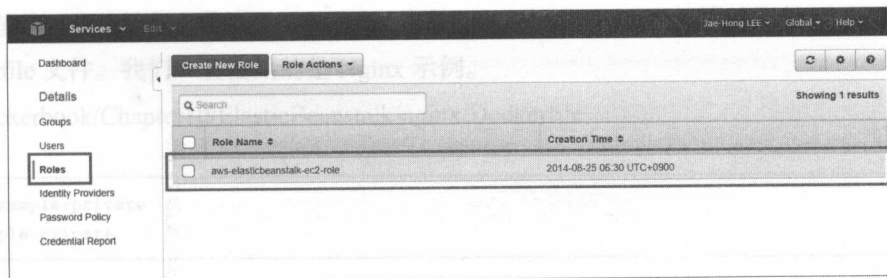


图 10-27 IAM 角色列表

转到 `aws-elasticbeanstalk-ec2-role` 详细设置页面。单击 `Attach Role Policy` 按钮。



图 10-28 IAM 角色详细设置

在 `Select Policy Template` 中单击 `Amazon S3 Read Only Access` 的 `Select` 按钮。只有拥有该权限，才能在 Elastic Beanstalk 中读取 S3 Bucket 中存储的 `.dockercfg` 文件。

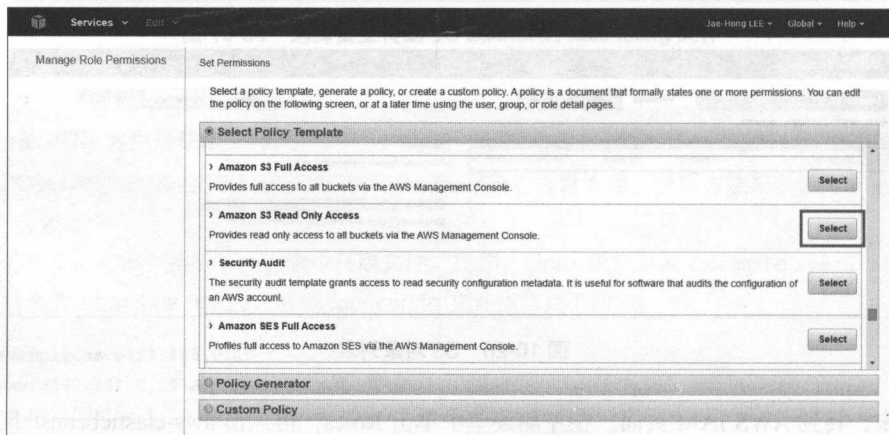


图 10-29 IAM 角色权限设置

显示设置 S3 读取权限的 Policy Document。单击 `Apply Policy` 按钮。

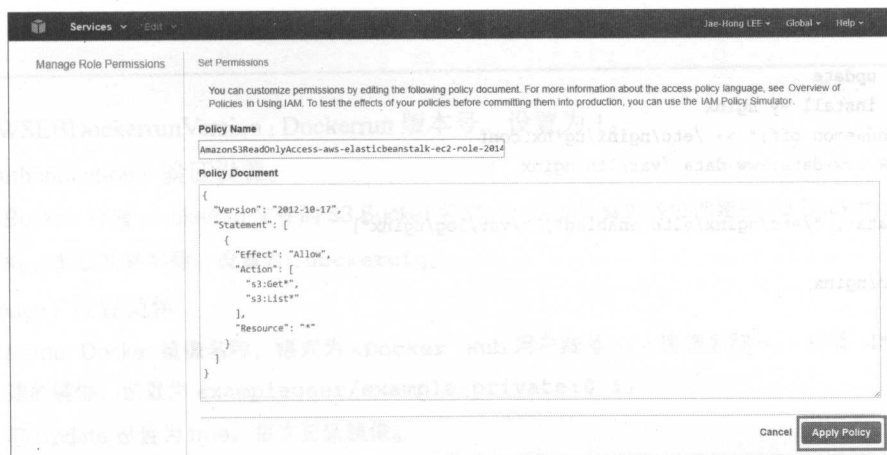


图 10-30 向 IAM 角色添加 S3 读权限

向 aws-elasticbeanstalk-ec2-role 添加 S3 读权限。

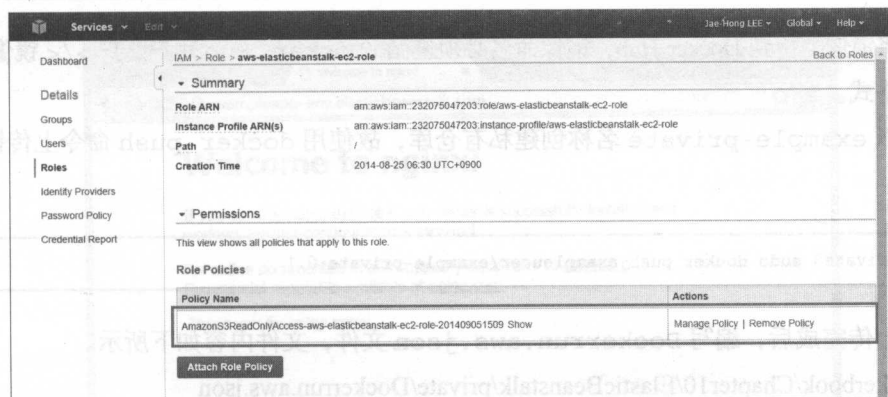


图 10-31 向 IAM 角色成功添加 S3 读取权限

请参考 13.3 节，使用 example-private 名称创建私有仓库。创建仓库后，使用 Dockerfile 文件创建镜像，再将镜像上传到 Docker Hub 私有仓库。首先创建 example-private 目录，编写 Dockerfile 文件。我们一直使用的是 Nginx 示例。

➤ dockerbook/Chapter10/ElasticBeanstalk/nginx/Dockerfile

```
~$ mkdir example-private
~$ cd example-private
```

```
> ~/example-private/Dockerfile
```

FROM ubuntu:14.04

```
MAINTAINER Foo Bar <exampleuser@example.com>
```

```
RUN apt-get update
```

```
RUN apt-get install -y nginx
```

```
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
```

```
RUN chown -R www-data:www-data /var/lib/nginx
```

```
VOLUME ["/data", "/etc/nginx/site-enabled", "/var/log/nginx"]
```

```
WORKDIR /etc/nginx
```

```
CMD ["nginx"]
```

```
EXPOSE 80
```

```
EXPOSE 443
```

使用 docker build 命令创建镜像。

```
~/example-private$ sudo docker build --tag exampleuser/example-private:0.1 .
```

若想将镜像上传到 Docker Hub，镜像命名必须遵循 <Docker Hub 用户账户>/<镜像名称>:<标签> 格式。

由于以 example-private 名称创建私有仓库，故使用 docker push 命令上传镜像，如下所示。

```
~/example-private$ sudo docker push exampleuser/example-private:0.1
```

镜像上传完成后，编写 Dockerrun.aws.json 文件，文件内容如下所示。

➤ dockerbook/Chapter10/ElasticBeanstalk/private/Dockerrun.aws.json

> Dockerrun.aws.json

```
{
  "AWSEBDockerrunVersion": "1",
  "Authentication": {
    "Bucket": "examplebucket10",
    "Key": ".dockercfg"
  },
  "Image": {
    "Name": "exampleuser/example-private:0.1",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "80"
    }
  ]
}
```

- » **AWSEBDockerrunVersion**: Dockerrun 版本号。设置为 1。
- » **Authentication**: 验证设置。
 - » **Bucket**: 存储 .dockercfg 文件的 S3 Bucket 名称。一定要设置为各位创建的 S3 Bucket 名称。
 - » **Key**: 验证文件名称。设置为 .dockercfg。
- » **Image**: 设置镜像。
 - » **Name**: Docker 镜像名称, 格式为 <Docker Hub 用户账号>/<镜像名称>:<标签>。根据前面创建的镜像, 设置为 exampleuser/example-private:0.1。
 - » 将 **Update** 设置为 true, 每次更新镜像。
- » **Ports**: 设置要与 Elastic Beanstalk 的 EC2 实例连接的容器端口号。

在 Elastic Beanstalk 环境页面单击 Upload and Deploy 按钮, 上传 Dockerrun.aws.json 文件后创建 Docker 容器。

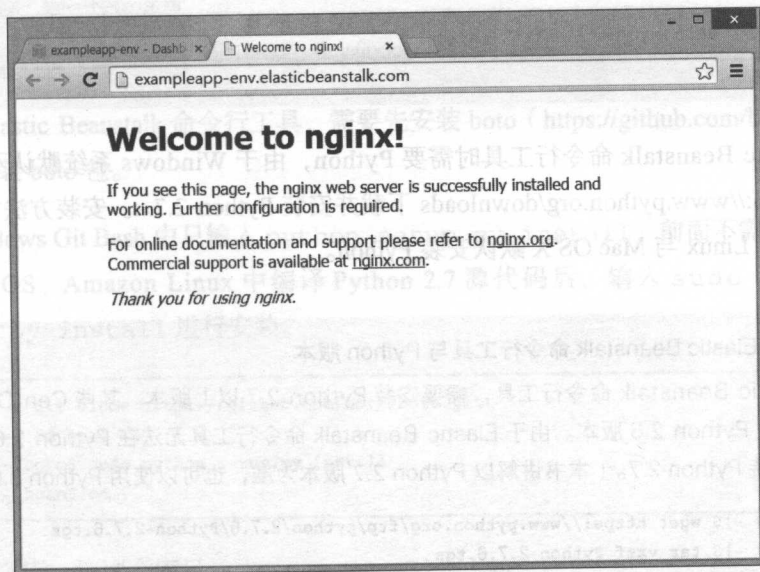


图 10-32 连接 Elastic Beanstalk 中使用私有仓库镜像创建的容器

10.2.4 使用 Git 部署 Elastic Beanstalk Docker 应用程序

下面使用 Git 在 Elastic Beanstalk 环境中部署 Elastic Beanstalk Docker 应用程序。

关于安装 Git 的方法, 请参考 8.1.1 节。

安装 Git 后, 需要运行 Elastic Beanstalk 命令行工具。在谷歌中搜索 AWS Elastic Beanstalk

Command Line Tool, 或者访问 <http://aws.amazon.com/code/6752709412171743>, 单击 Download 按钮, 下载 AWS-ElasticBeanstalk-CLI-2.6.3.zip 文件。(各位实际下载到的可能是更新的版本。)

将 AWS-ElasticBeanstalk-CLI-2.6.3.zip 文件下载到用户主目录 (如 /home/pyrasis), 然后解压缩。(对于 Windows 与 Mac OS X, 在浏览器与 Finder 中解压 zip 文件。)

> Linux

```
pyrasis@ubuntu:~$ wget https://s3.amazonaws.com/elasticbeanstalk/cli/AWS-ElasticBeanstalk-CLI-2.6.3.zip
pyrasis@ubuntu:~$ unzip AWS-ElasticBeanstalk-CLI-2.6.3.zip
```

提示 安装 unzip

若无 unzip 命令, 则需要先安装。

> CentOS

```
[pyrasis@centos ~]$ sudo yum install unzip
```

> Ubuntu

```
pyrasis@ubuntu:~$ sudo apt-get install unzip
```

运行 Elastic Beanstalk 命令行工具时需要 Python, 由于 Windows 系统默认不安装 Python, 所以先要从 <http://www.python.org/downloads> 下载并安装 Python 2.7.x。安装方法没有特别之处, 不再另行说明。Linux 与 Mac OS X 默认安装 Python。

提示 Linux 中 Elastic Beanstalk 命令行工具与 Python 版本

若想使用 Elastic Beanstalk 命令行工具, 需要安装 Python 2.7 以上版本。某些 CentOS 与 Amazon Linux 安装的是 Python 2.6 版本。由于 Elastic Beanstalk 命令行工具无法在 Python 2.6 版本中使用, 所以需要先安装 Python 2.7。(本书讲解以 Python 2.7 版本为准, 也可以使用 Python 3.0 以上版本。)

```
[pyrasis@centos ~]$ wget https://www.python.org/ftp/python/2.7.6/Python-2.7.6.tgz
[pyrasis@centos ~]$ tar vxzf Python-2.7.6.tgz
[pyrasis@centos ~]$ cd Python-2.7.6
[pyrasis@centos Python-2.7.6]$ ./configure
[pyrasis@centos Python-2.7.6]$ sudo make; sudo make install
```

接下来, 在 Windows 中运行 Git Bash。(在 Git Bash 中可以运行 Unix/Linux 方式的命令, 所以从现在起出现的命令在 Mac OS X、Linux、Windows 中都是通用的。)可以直接在 Linux 与 Mac OS X 的终端中运行。



图 10-33 在 Windows 运行 Git Bash

要运行 Elastic Beanstalk 命令行工具，需要先安装 boto (<https://github.com/boto/boto>)。输入如下命令，安装 boto 包。

- 在 Windows Git Bash 中只输入 `python setup.py install`，前面不需要输入 `sudo`。
- 在 CentOS、Amazon Linux 中编译 Python 2.7 源代码后，输入 `sudo python 2.7 setup.py install` 进行安装。

```
pyrasis@ubuntu:~$ git clone https://github.com/boto/boto.git
pyrasis@ubuntu:~$ cd boto
pyrasis@ubuntu:~/boto$ sudo python setup.py install
pyrasis@ubuntu:~/boto$ cd ..
```

创建 Git 仓库，转到仓库目录。

```
~$ git init exampleapp
~$ cd exampleapp
```

向刚刚创建的 Git 仓库安装 `aws.push` 命令，Windows 的 Git Bash 中可以直接安装 `AWSDevTools-RepositorySetup.sh`。

Linux

```
pyrasis@ubuntu:~/exampleapp$ ../AWS-ElasticBeanstalk-CLI-2.6.3/AWSDevTools/Linux/AWSDevTools-RepositorySetup.sh
```

向 Git 安装 AWS 访问密钥与秘密密钥。

- AWS Access key：输入访问密钥。
- AWS Secret key：输入秘密密钥。
- AWS Region：输入地区简称。我输入 ap-northeast-1，即 Tokyo 地区。
- AWS Elastic Beanstalk Application：输入 Elastic Beanstalk 应用程序名称。输入前面创建的 exampleapp。
- AWS Elastic Beanstalk Environment：输入 Elastic Beanstalk 环境名称。输入前面创建的 exampleapp-env。

```
pyrasis@ubuntu:~/exampleapp$ git aws.config
AWS Access Key: AKIAJMV5Z7VUHUCJJFNA
AWS Secret Key: Sg1k7MeAZKKJbrtMKEdmF6wfoROqUA4kT+Im9OED
AWS Region [default to us-east-1]: ap-northeast-1
AWS Elastic Beanstalk Application: exampleapp
AWS Elastic Beanstalk Environment: exampleapp-env
```

接下来，编写一个简单的 Web 页面。将下面内容保存为 app.js 文件。

- dockerbook/Chapter10/ElasticBeanstalk/git/app.js

> app.js

```
var express = require('express');
var app = express();

app.get(['/', '/index.html'], function (req, res) {
  res.send('Hello Docker - Git');
});

app.listen(80);
```

为了使用 Node.js npm 包，编写如下代码，然后保存为 package.json 文件。

- dockerbook/Chapter10/ElasticBeanstalk/git/package.json

> package.json

```
{
  "name": "hello",
  "description": "Hello Elastic Beanstalk",
```

```
"version": "0.0.1",
"dependencies": {
  "express": "4.4.x"
}
}
```

将如下内容保存到 Dockerfile 文件。

» `dockerbook/Chapter10/ElasticBeanstalk/git/package.json`

> Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y nodejs npm

ADD app.js /var/www/app.js
ADD package.json /var/www/package.json

WORKDIR /var/www
RUN npm install

CMD nodejs app.js
EXPOSE 80
```

将 `app.js`、`package.json`、`Dockerfile` 文件提交到 Git 仓库。

```
pyrasis@ubuntu:~/exampleapp$ git add app.js
pyrasis@ubuntu:~/exampleapp$ git add package.json
pyrasis@ubuntu:~/exampleapp$ git add Dockerfile
pyrasis@ubuntu:~/exampleapp$ git commit -m "Hello Elastic Beanstalk"
```

接着，使用 `aws.push` 命令将应用程序源代码上传到 Elastic Beanstalk。

```
pyrasis@ubuntu:~/exampleapp$ git aws.push
```

转到 Elastic Beanstalk 环境页面。短暂等待后，Health 由 Updating 变为 Green，Elastic Beanstalk 应用程序部署完成。Running Version 中出现 `git-<Revision>` 字样。点击上方的 <环境名称>.`elasticbeanstalk.com` 链接。

```
$ curl https://sdk.cloud.google.com | bash
$ source ~/.bashrc
```

首先使用 `gcloud` 命令登录谷歌云。使用 `gcloud` 显示如下验证 URL。

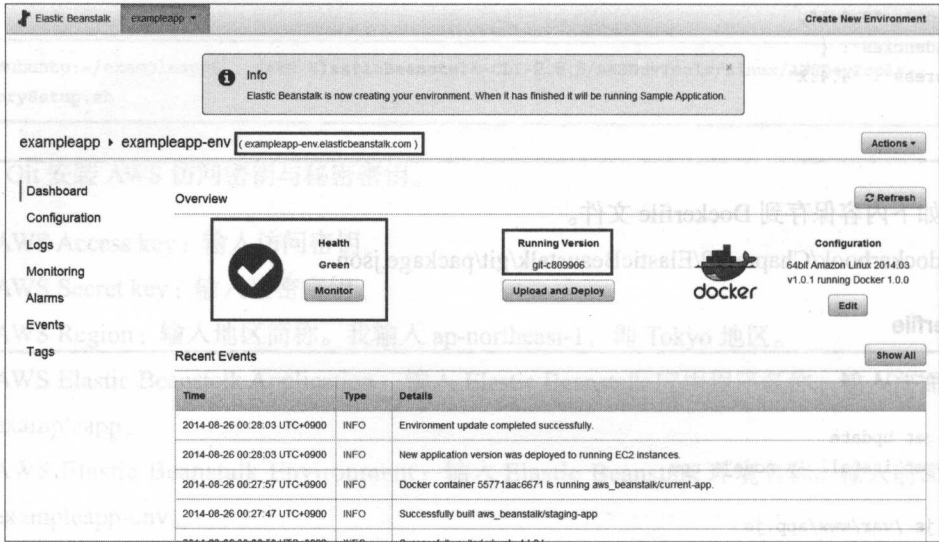


图 10-34 使用 Git 部署 Elastic Beanstalk 应用程序

从 Web 浏览器访问 Elastic Beanstalk URL，显示 app.js 的内容。

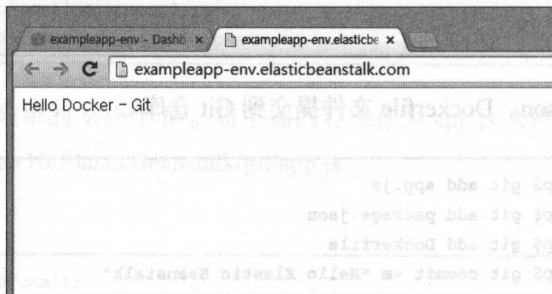


图 10-35 连接 Elastic Beanstalk Docker 应用程序

第 11 章

DOCKER

在 Google Cloud Platform 中使用 Docker

本章将学习如何在 Google Cloud Platform 的 Compute Engine 与 Container Engine 中使用 Docker。

请注意，必须拥有谷歌账户才能使用 Compute Engine 与 Container Engine。若没有谷歌账户，请先申请。虽然 Google Cloud Platform 是付费服务，但首次加入将有 60 天免费试用机会（价值 300 美元）。

提示 ▶ 加入 Google Cloud Platform 与海外验证 1 美元

加入 Google Cloud Platform 时，要验证信用卡是否真实有效，所以会暂时从卡中扣除 1 美元。信用卡通过验证后，会自动返还扣除的 1 美元。

11.1 ▶ 安装 Google Cloud SDK

从下列 URL 地址下载 Google Cloud SDK。

▶ <https://cloud.google.com/sdk/>

若使用 Windows 系统，请直接下载并安装 GoogleCloudSDKInstaller.exe。而 Mac OS X 与 Linux 中，要在终端执行如下命令。虽然出现多种选择项，但保持所有默认设置不变。

```
$ curl https://sdk.cloud.google.com | bash
$ source ~/.bashrc
```

首先使用 gcloud 命令登录谷歌云。使用 gcloud 时，显示如下验证 URL。

```
$ gcloud auth login
```

Your browser has been opened to visit:

```
https://accounts.google.com/o/oauth2/auth?redirect_uri=http%3A%2F%2Flocalhost%3A8085%2F&prompt=select_account&response_type=code&client_id=32555940559.apps.googleusercontent.com&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcloud-platform+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fappengine.admin+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcompute&access_type=offline
```

运行该命令将直接打开默认 Web 浏览器，并连接验证 URL。出现账户选择画面时，选择加入 Google Cloud Platform 的谷歌账户。

出现请求 Google Cloud SDK 权限画面，单击“允许”按钮。



图 11-1 选择 Google 账户



图 11-2 请求 Google Cloud SDK 权限

显示认证完成页面。

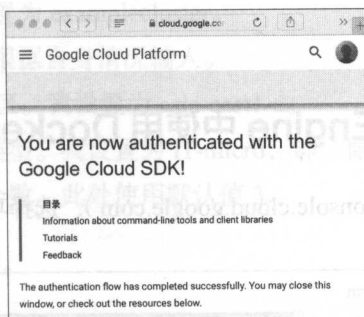


图 11-3 Google Cloud SDK 认证完成

也可以通过该 URL <https://cloud.google.com/sdk/docs/quickstarts> 查看不同平台的快速入门文档，根据说明逐步操作。

11.2 在 Compute Engine 中使用 Docker

由于谷歌开发者控制台中仍然无法使用容器镜像，所以要通过 Google Cloud SDK 创建 VM 实例。

使用 `gcloud compute instances create` 命令创建 VM 实例。该命令的具体说明参见 URL <https://cloud.google.com/sdk/gcloud/reference/compute/instances/create>。

```
$ gcloud compute instances create example-docker \
  --image container-vm \
  --image-project google-containers \
  --zone asia-east1-a \
  --machine-type f1-micro
```

- 设置 VM 实例名称为 `example-docker`。
- `--image`：镜像名称。通过该命令查询可用镜像的版本信息。
`gcloud compute images list --project google-containers`
- `--image-project`：设置 `google-containers`。
- `--zone`：要创建 VM 实例的区域，我设置为 `asia-east1-a`。
- `--machine-type`：VM 实例机类型。我设置为 `f1-micro`。

创建 VM 实例后，显示到谷歌开发者控制台的“计算（Computing）→计算引擎（Compute Engine）→VM 实例”页面。单击 `example-docker` 镜像的 SSH 按钮。

在 Web 浏览器中使用 SSH 连接 `example-docker` 实例。现在开始可以使用 Docker 了。

也可以使用 Google Cloud SDK 通过 SSH 进行连接, 命令格式为 `gcloud compute ssh --zone< 地区名称 > 实例名称 >`。

11.3 在 Container Engine 中使用 Docker

连接谷歌云控制台 (<https://console.cloud.google.com>), 选择项目。可以选择“容器引擎”。



图 11-4 选择“容器引擎”



图 11-5 准备创建容器集群

点击“创建容器集群”。

- 名称: 集群名称。此处设置为 examplecluster。
- 说明: 集群说明。请各位根据自身情况输入。
- 地区: 创建 VM 实例的地区。我设置为 asia-east1-a。
- 机器类型: VM 实例机器类型。我设置为 f1-micro, 即“微型(1 个共享 vCPU)”。
- 集群大小: VM 实例集群个数。此处使用默认值 3。

设置完成后, 单击创建按钮。

Google Cloud-Platform

容器引擎

← 创建容器集群

容器集群是由相同 VM 组成的一个用于运行 Kubernetes 的受管群组 了解详情

名称 @

examplecluster

说明 (可选)

地区 @

asia-east1-a

机器类型

1 个 vCPU 3.75 GB 内存 自定义

集群大小 @

3

总核心数 3 个 vCPU

总内存 11.25 GB

集群实例使用临时本地磁盘。如果需要, 您可以向容器组附加一个永久磁盘。

子网 @

default:3ee9eb5ff31cd1c

日志与监控 @

☐ 开启云监控

要使用云监控来监测实例, 请为项目启用云监控 了解详情

☒ 开启云日志

展开

您需要为集群中的 3 个节点 (VM 实例) 付费。了解详情

创建 取消

等效 REST 命令行

图 11-6 创建容器集群

容器集群创建完毕。

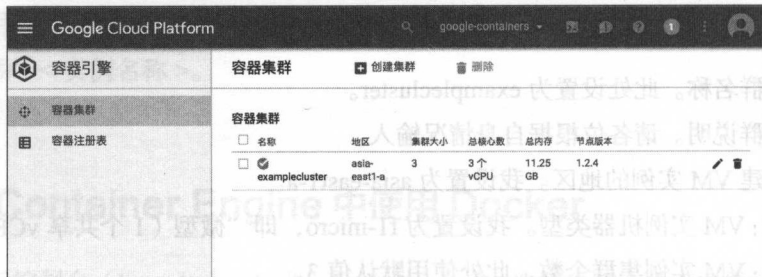


图 11-7 容器集群创建完毕

打开 Google 云控制台，查看“计算引擎”界面，点击“VM 实例”，可以看到我们刚刚创建的 3 个实例。



图 11-8 容器集群的 3 个实例

可以看到，Google 的容器引擎是使用 Kubernetes 管理的。感兴趣的读者可以访问 URL <https://cloud.google.com/container-engine/docs/quickstart>，运行示例的“Hello Node”例子。

提示 Kubernetes

Kubernetes 是谷歌开发的容器、集群管理工具。

➤ <https://github.com/GoogleCloudPlatform/kubernetes>

第 12 章

DOCK E R

使用 Docker Hub

Docker Hub 网站提供 Docker 官方镜像，用户之间共享镜像。

- ▶ 公共仓库 (Public Repository)：所有人均可使用公共仓库中存储的镜像。用户可以免费创建公共仓库，且个数不受限制。
- ▶ 私有仓库 (Private Repository)：私有仓库中存储的镜像仅供个人使用，用户只能免费创建 1 个私有仓库，创建更多私有仓库需要额外付费。
- ▶ 收藏夹 (Starred)：用户可以将别人有用的仓库添加到收藏夹，以便下次使用。
- ▶ Automated Build：该功能与 GitHub、BitBucket 仓库联动，推送 Dockerfile 时自动构建镜像。

12.1 ▶ 加入 Docker Hub

要使用 Docker Hub，必须先加入。运行 Web 浏览器，访问 <https://hub.docker.com>。

输入 ID、密码、电子邮箱，然后单击 Sign up 按钮。若有 GitHub 账户，也可以使用 GitHub 账户直接加入。

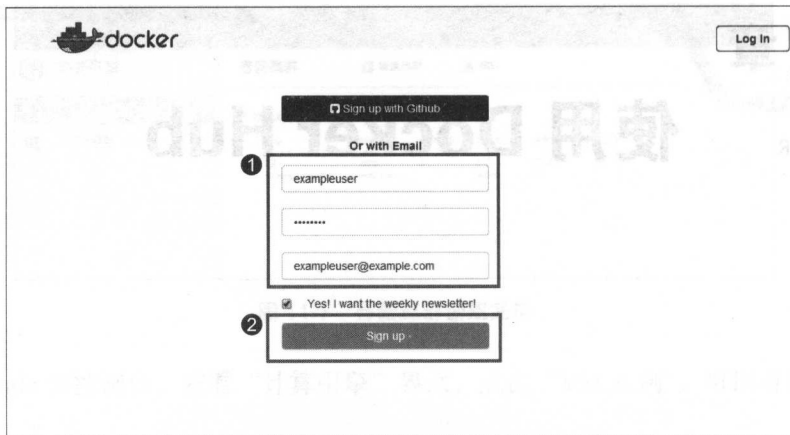


图 12-1 加入 Docker Hub

Docker Hub 加入完成。

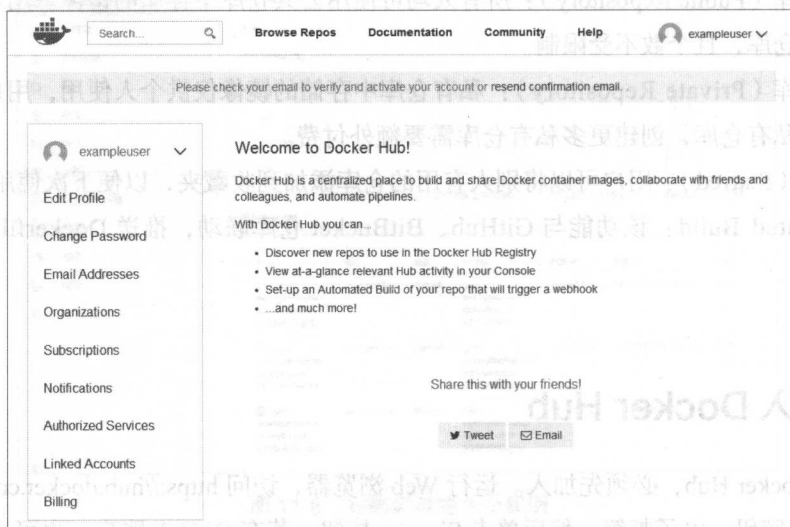


图 12-2 Docker Hub 加入完成

进入加入时输入的电子邮箱，打开验证邮件，单击 Confirm Your Email 按钮。



图 12-3 Docker Hub 验证邮件

电子邮件验证完成。

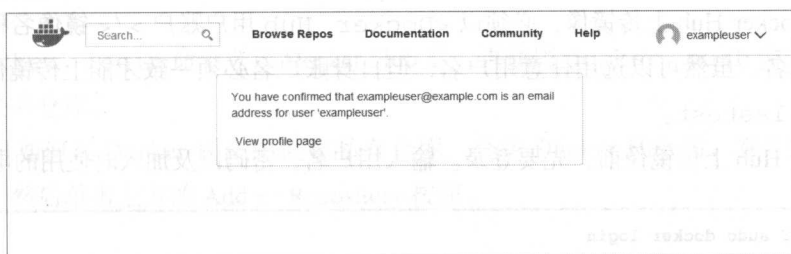


图 12-4 Docker Hub 电子邮件验证完成

12.2 ▶ 使用 push 命令上传镜像

下面向 Docker Hub 公共仓库上传镜像。首先创建 `example-nginx` 目录，生成 Dockerfile 文件。以一直使用的 Nginx 为例。

▶ `dockerbook/Chapter13/Dockerfile`

```
~$ mkdir example-nginx
~$ cd example-nginx
```

```
> ~/example-nginx/Dockerfile
```

```
FROM ubuntu:14.04
```



```

MAINTAINER Foo Bar <exampleuser@example.com>

RUN apt-get update
RUN apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
RUN chown -R www-data:www-data /var/lib/nginx

VOLUME ["/data", "/etc/nginx/site-enabled", "/var/log/nginx"]

WORKDIR /etc/nginx

CMD ["nginx"]

EXPOSE 80
EXPOSE 443

```

使用 docker build 命令创建镜像。

```
~/example-nginx$ sudo docker build --tag exampleuser/example-nginx:0.1 .
```

若要向 Docker Hub 上传镜像，必须以 <Docker Hub 用户账户>/<镜像名称>:<标签> 格式生成镜像名。虽然可以选用任意用户名，但自身账户名必须一致才能上传镜像。若不设置标签，则使用 latest。

向 Docker Hub 上传镜像前，先要登录。输入用户名、密码以及加入时使用的电子邮箱。

```
~/example-nginx$ sudo docker login
Username: exampleuser
Password:
Email: exampleuser@example.com
Login Succeeded
```

接下来，使用 docker push 命令上传镜像。

```
~/example-nginx$ sudo docker push exampleuser/example-nginx:0.1
```

命令格式为 docker push <Docker Hub 用户账号>/<镜像名称>:<标签>。

短暂等待后，镜像完全上传到 Docker Hub 公共仓库。Docker Hub 页面中，单击 Repositories 菜单，显示刚刚上传的 <Docker Hub 用户账号>/example-nginx 镜像。（若 Docker Hub 中不存在创建的公共仓库，则上传镜像时会自动创建公共仓库。）

也可以在图 12-5 中先通过 Add Repository 按钮创建公共仓库，然后使用 docker push 命令上传镜像。这样，其他人就可以使用 docker pull <Docker Hub 用户账号>/example-nginx 命令下载并使用 example-nginx 镜像。

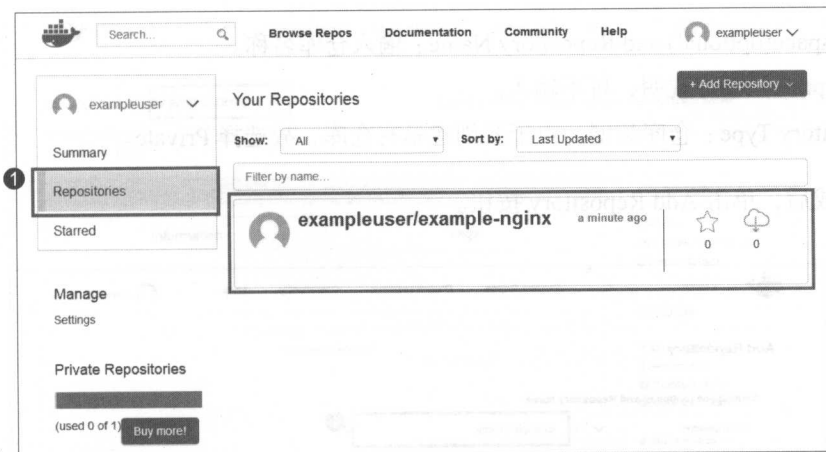


图 12-5 向 Docker Hub 仓库上传镜像成功

12.3 ▶ 创建 Docker Hub 私有仓库

使用 `docker login` 命令登录 Docker Hub 后，再使用 `docker push` 命令上传镜像时，将自动创建公共仓库。

下面学习如何在 Docker Hub 中创建私有仓库。登录 Docker Hub 后，在左侧菜单中单击 **Repositories**，然后单击上方的 **Add → Repository** 按钮。

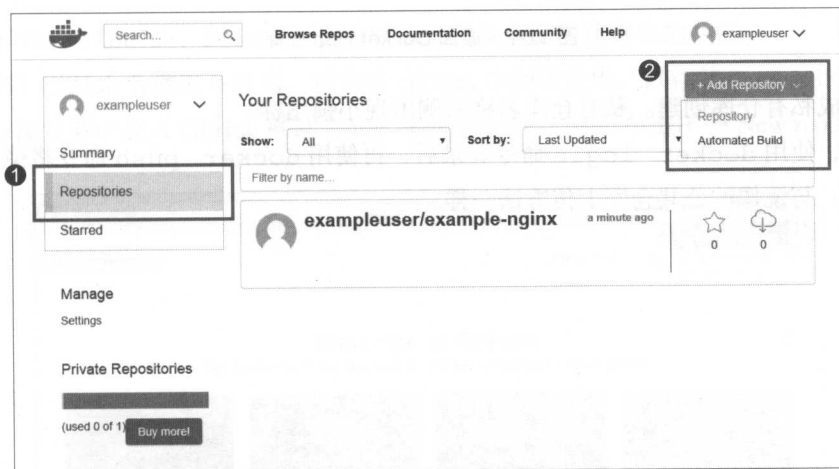


图 12-6 在 Docker Hub 仓库目录创建仓库

添加仓库。如图 12-7 所示。

- Namespace(optional) and Repository Name：输入仓库名称。
- Description：仓库说明，可不输入。
- Repository Type：仓库类型。由于要创建私有仓库，故选择 Private。

设置完成后，单击 Add Repository 按钮。

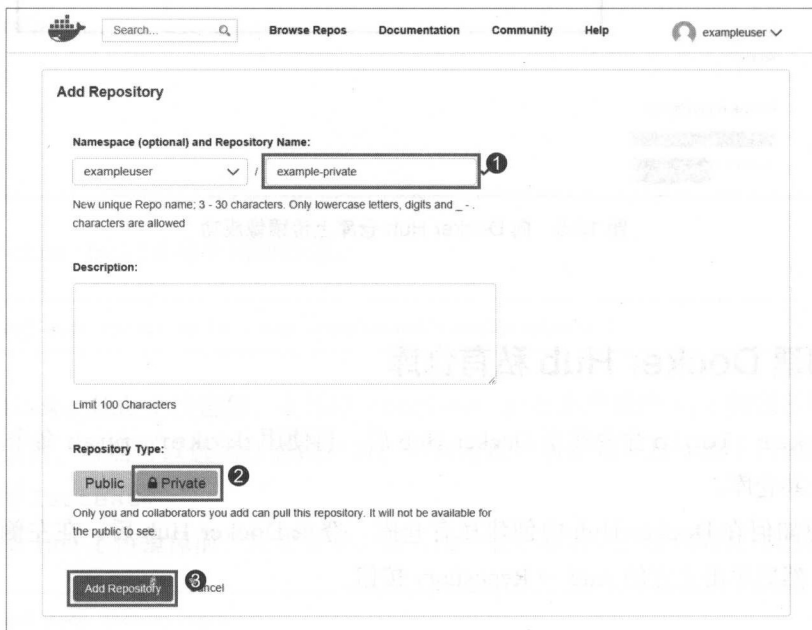


图 12-7 添加 Docker Hub 仓库

至此完成私有仓库创建。私有仓库名称右侧出现小锁图标。

接下来，使用 `docker login` 命令登录后，再使用 `docker push` 命令将镜像上传到私有仓库即可。与镜像的公共仓库上传方法一样。

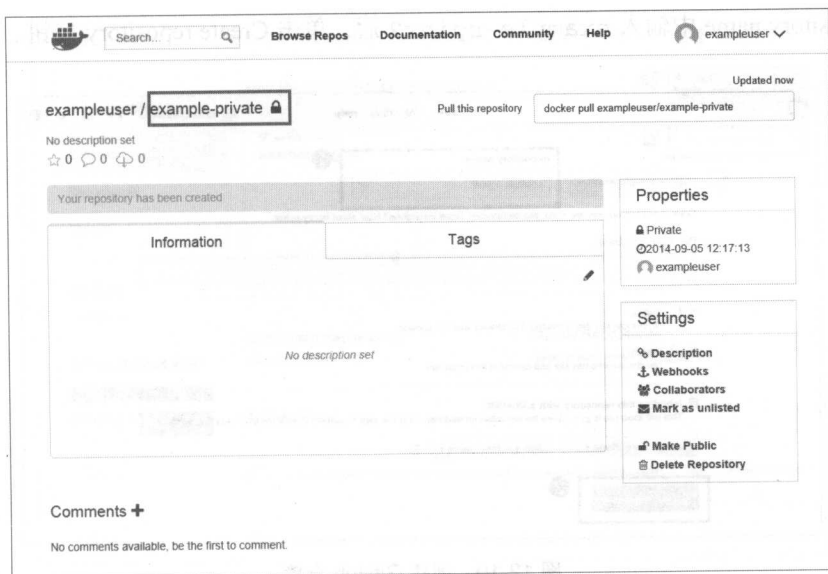


图 12-8 Docker Hub 私有仓库创建完成

```
~/example-private$ sudo docker push exampleuser/example-private:0.1
```

12.4 ▸ 使用 Docker Hub Automated Build

Docker Hub 与 GitHub、BitBucket 保持联动，提供自动构建镜像的功能。

本书将以 GitHub 为例进行说明。首先在 GitHub 中创建仓库，以存储 Dockerfile 文件。在 Web 浏览器中进入 GitHub 网站 (<http://github.com>)，然后单击 + → New repository。



图 12-9 创建 GitHub 仓库

在 Repository name 中输入 example-nginx2 后，单击 Create repository 按钮。

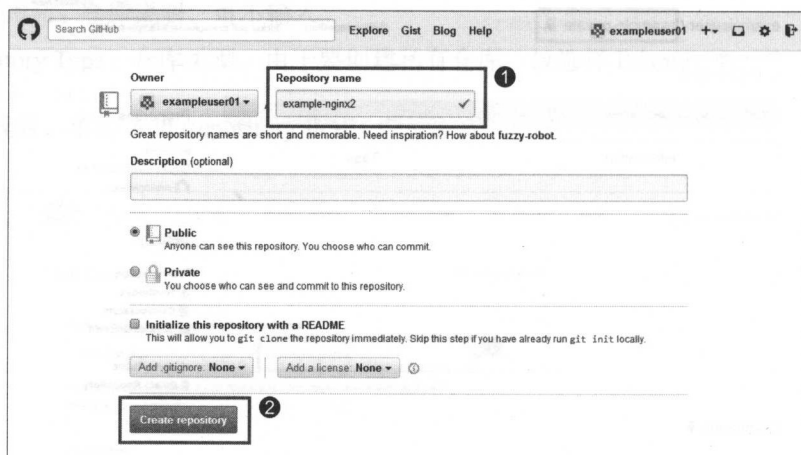


图 12-10 创建 GitHub 仓库

GitHub 仓库创建完成，仓库地址格式为 `git@github.com:<GitHub 用户账号>/example-nginx2.git`。

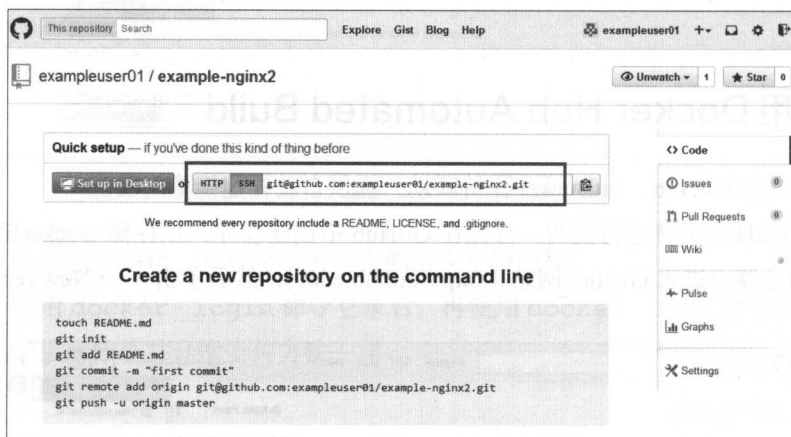


图 12-11 GitHub 仓库创建完成

转到 Docker Hub (<https://hub.docker.com>) 后，单击上方 Add Repository → Automated Build。

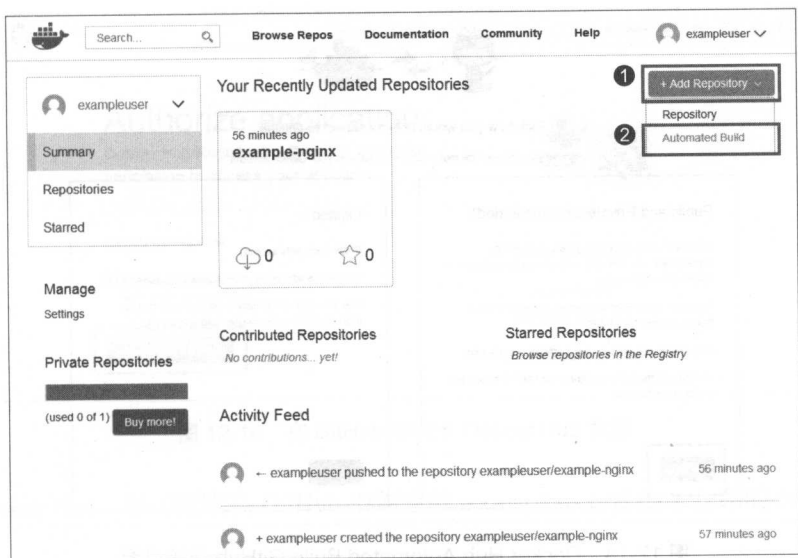


图 12-12 创建 Docker Hub Automated Build

可以使用 GitHub 与 BitBucket。由于前面创建的是 GitHub 仓库，所以单击 GitHub 的 Select 按钮。

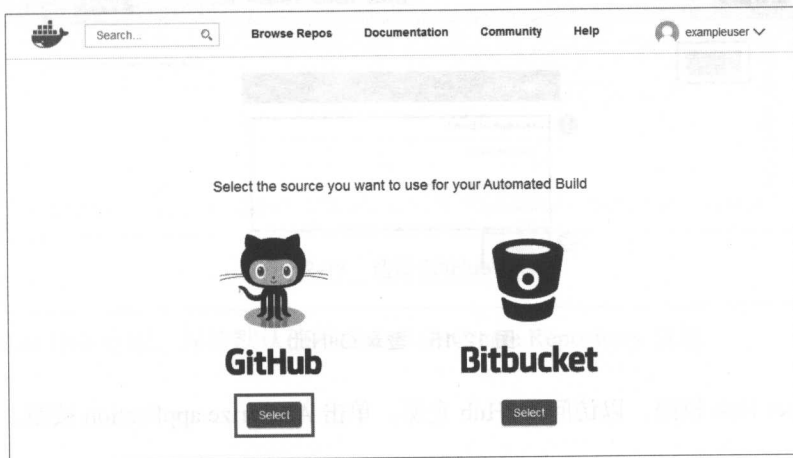


图 12-13 选择 Docker Hub Automated Build 源码仓库网站

接下来是 GitHub 连接设置。单击 Public and Private 的 Select 按钮。

- **Public and Private**：可以使用公共、私有仓库以及个体组织（Organization），自动设置为 Service Hook。
- **Limited**：只能使用公共仓库与组织，必须手动设置。

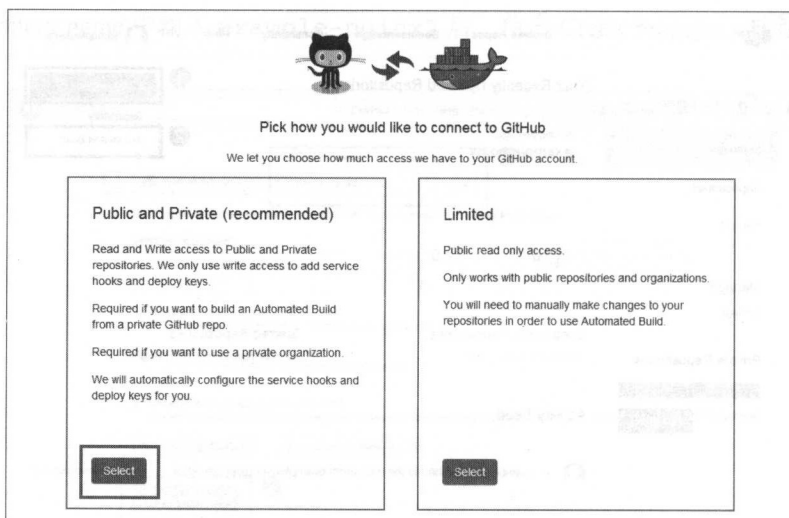


图 12-14 Docker Hub Automated Build GitHub 连接设置

若尚未登录 GitHub，将显示登录界面，使用 GitHub 账户登录。若无 GitHub 账户，请先加入 GitHub。（加入方法不再另行说明。）

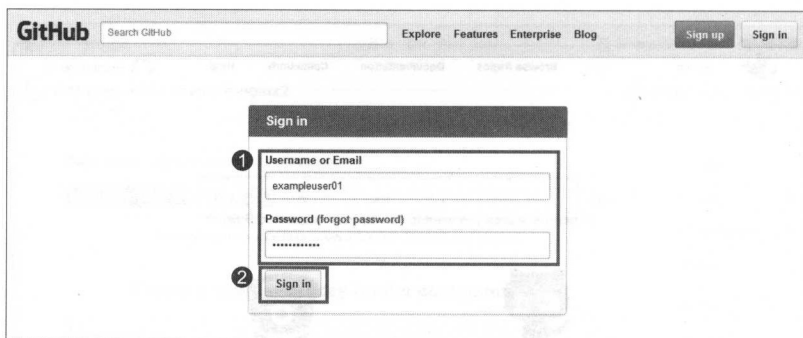


图 12-15 登录 GitHub

赋予 Docker Hub 权限，以访问 GitHub 仓库。单击 Authorize application 按钮。

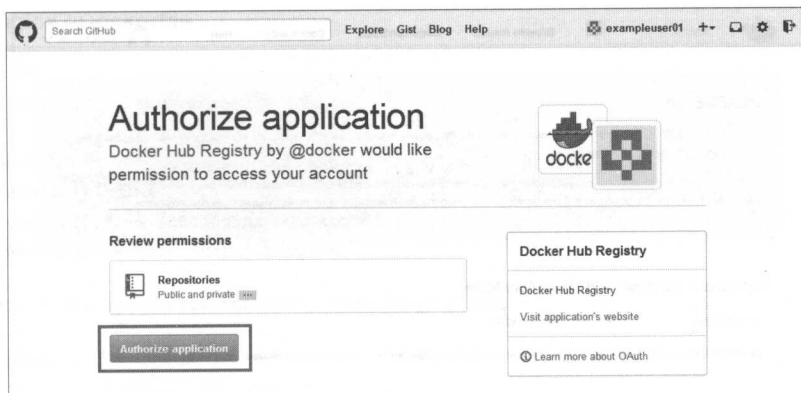


图 12-16 在 GitHub 中赋予 Docker Hub 权限

返回 Docker Hub, 单击刚刚在 GitHub 中创建的 example-nginx2 仓库的 Select 按钮。

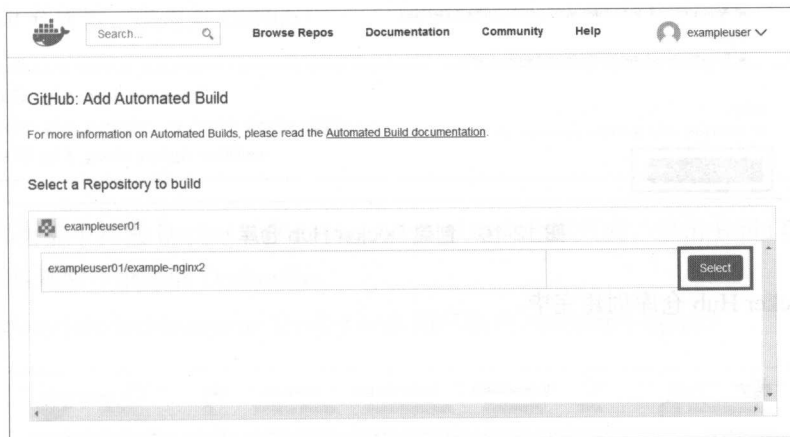
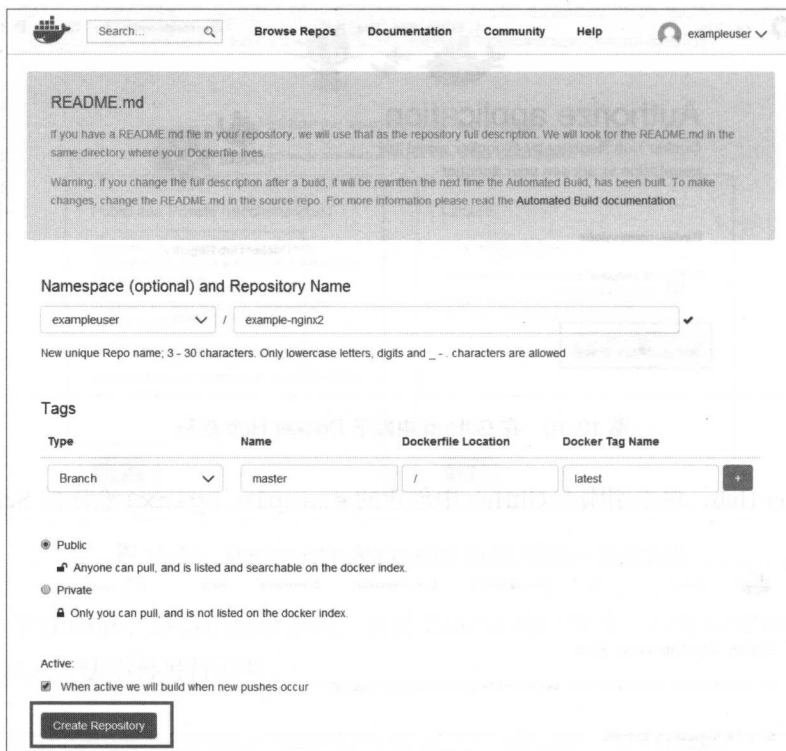


图 12-17 选择 GitHub 仓库

创建 Docker Hub 仓库。保持默认值不变, 单击 Create Repository 按钮。



The screenshot shows the Docker Hub 'Create Repository' form. At the top, there's a navigation bar with 'Search...', 'Browse Repos', 'Documentation', 'Community', 'Help', and a user profile 'exampleuser'. Below this is a 'README.md' section with instructions on how Docker Hub uses the README file. The main form area is titled 'Namespace (optional) and Repository Name' and contains two input fields: 'exampleuser' and 'example-nginx2'. Below these fields is a note: 'New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and _ - . characters are allowed'. The 'Tags' section has a table with columns: 'Type', 'Name', 'Dockerfile Location', and 'Docker Tag Name'. The first row has 'Branch' as the type, 'master' as the name, and 'latest' as the tag name. Below the table are radio buttons for 'Public' (selected) and 'Private'. The 'Active' section has a checked checkbox 'When active we will build when new pushes occur'. At the bottom is a 'Create Repository' button.

图 12-18 创建 Docker Hub 仓库

至此，Docker Hub 仓库创建完毕。



The screenshot shows the 'What's Next' page on Docker Hub after repository creation. It features a message: 'Configuration saved, and a build was triggered for exampleuser01/example-nginx2. Check back in a few minutes for the results!'. Below this is a 'What's Next' section with instructions: 'You have successfully configured a Automated Build with Github repo exampleuser01/example-nginx2. Visit your [build details](#) page, to track your builds. Make sure your Automated Build builds correctly. If it doesn't, look at the error logs to see what is causing your problem. If you have any questions or issues, please let us know.'

图 12-19 Docker Hub 仓库创建完毕

下载前面创建的 GitHub 仓库，然后转到仓库目录。

```
~$ git clone git@github.com:<GitHub 用户账户>/example-nginx2.git
~$ cd example-nginx2
```

将如下内容保存为 Dockerfile 文件，放到 Git 仓库目录。

► dockerbook/Chapter13/Dockerfile

> ~/example-nginx2/Dockerfile

```

FROM ubuntu:14.04
MAINTAINER Foo Bar <exampleuser@example.com>

RUN apt-get update
RUN apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
RUN chown -R www-data:www-data /var/lib/nginx

VOLUME ["/data", "/etc/nginx/site-enabled", "/var/log/nginx"]

WORKDIR /etc/nginx

CMD ["nginx"]

EXPOSE 80
EXPOSE 443

```

提交到 Git 仓库后，推送至 GitHub。

```

~/example-nginx2$ git add Dockerfile
~/example-nginx2$ git commit -m "add Dockerfile"
~/example-nginx2$ git push origin master

```

在 Web 浏览器中转到 Docker Hub example-nginx2 仓库页面。单击 Build Details 选项卡，可以看到正在构建刚才推送的 Dockerfile。

<https://registry.hub.docker.com/u/<Docker Hub 用户账户>/example-nginx2>

The screenshot shows the Docker Hub interface for the repository 'exampleuser / example-nginx2'. The 'Build Details' tab is active, displaying a table of build history. The table has columns for 'build id', 'Status', 'Created Date', and 'Last Updated'. The first build, with id 'bvtalqrtg2ar9ucmx4dpw', is in 'Building' status and was created on 2014-08-28 06:35:56. The second build, with id 'tmeov4hs8xsazmgbnwjdw', is in 'Error' status and was created on 2014-08-28 06:17:25. The interface also shows tabs for 'Information', 'Dockerfile', 'Tags', and 'Build Details', along with a 'Start a Build' button and a 'Pull this repository' button.

build id	Status	Created Date	Last Updated
bvtalqrtg2ar9ucmx4dpw	Building	2014-08-28 06:35:56	2014-08-28 06:35:58
tmeov4hs8xsazmgbnwjdw	Error	2014-08-28 06:17:25	2014-08-28 06:17:27

图 12-20 Docker Hub Automated Build 正在运转

短暂等待后, Status 变为 Finished。每次向 GitHub 执行提交动作时, Docker Hub 中都会生成新镜像。

接下来, 设置 Git 标签与 Docker 镜像标签。首先在 Git 仓库中设置标签, 然后推送。

```
~/example-nginx2$ git tag -a 0.1 -m "Version 0.1"
~/example-nginx2$ git push origin 0.1
```

在 Docker Hub example-nginx2 仓库的 Build Details 选项卡中, 单击 Edit Build Details。

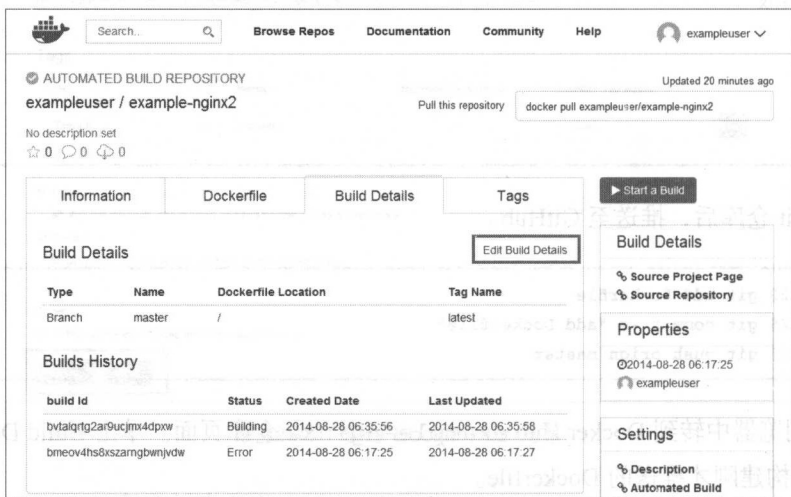


图 12-21 设置 Docker Hub Automated Build

单击 + 按钮, 添加设置。

- Type: Git 分支格式, 可以选择 Branch 与 Tag。此处选择 Tag。
- Name: Git 分支标签名称。输入 0.1。
- Dockerfile Location: 仓库中 Dockerfile 所在路径。使用默认值。
- Docker Tag Name: Docker 标签名称。输入 0.1。

设置完毕后, 单击 Save and trigger build。

向 Build Details 选项卡添加标签 0.1。若修改构建设置, 则再次构建所有标签, 所以构建 latest 与 0.1 两个标签, 如图 12-23 所示。

Search... Browse Repos Documentation Community Help exampleuser

Edit Automated Build settings for exampleuser01/example-nginx2

Repo Name:
exampleuser/example-nginx2

You can't change the Repo Name after the build has been created

Active:
☒

When active we will build when new pushes occur

Type	Name	Dockerfile Location	Docker Tag Name
Branch	master	/	latest
Tag	0.1	/	0.1

Save and trigger build

图 12-22 设置 Docker Hub Automated Build 标签

Search... Browse Repos Documentation Community Help exampleuser

Automated Build Repository exampleuser / example-nginx2 Updated 13 minutes ago

Pull this repository docker pull exampleuser/example-nginx2

No description set

Configuration saved, and a build was triggered for exampleuser01/example-nginx2. Check back in a few minutes for the results!

Start a Build

Type	Name	Dockerfile Location	Tag Name
Tag	0.1	/	0.1
Branch	master	/	latest

Builds History

build id	Status	Created Date	Last Updated
bdhxgnu5snjjsbcmvgnvpe	Building	2014-08-28 06:51:41	2014-08-28 06:54:10
bmyvsrftpb68mbspwupwd	Building	2014-08-28 06:51:40	2014-08-28 06:54:30
bvtalqrg2ar9ucjmx4dpw	Finished	2014-08-28 06:35:56	2014-08-28 06:38:47
bmeov4hs8xszamgbwnjydw	Error	2014-08-28 06:17:25	2014-08-28 06:17:27

Build Details

- Source Project Page
- Source Repository

Properties

- 2014-08-28 06:17:25
- exampleuser

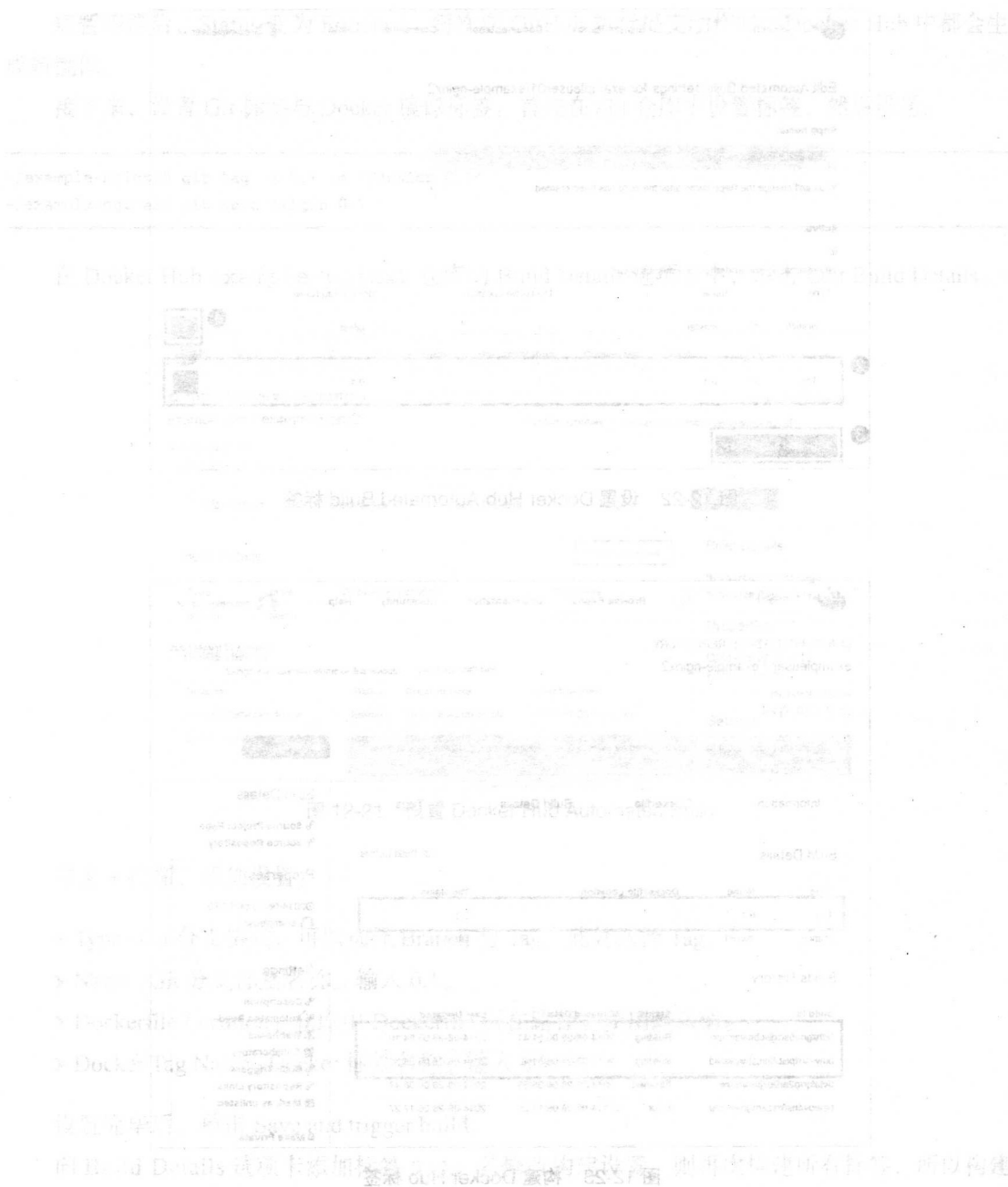
Settings

- Description
- Automated Build
- Webhooks
- Collaborators
- Build Triggers
- Repository Links
- Mark as unlisted
- Make Private

图 12-23 构建 Docker Hub 标签

短暂等待后，Status 变为 Finished，example-nginx2:0.1 镜像也可用。

以后若修改 Dockerfile 文件并推送分支、标签到 Git，则会新建相应镜像；而推送未在 Build Details 中设置的 Git 分支、标签时，将不会创建镜像。



如果等待，状态变为 finished，example/nginx:0.2 镜像也可用。以后若修改 Dockerfile 文件中构建分支，标签到 Git，则重新构建镜像，而推送至 Build Details 中设置的 Git 分支，标签时，将不创建镜像。

第 13 章

DOCKER

使用 Docker Remote API

Docker 守护进程与客户端通信时，若在本机，则使用 Unix 套接字；若在远程，则使用 TCP 套接字。此处，API 采用 HTTP REST 格式实现，所以并不从属于特定语言，可以在多种语言中使用。

首先停止 Docker 守护进程，再使用 TCP 套接字运行，以测试 API。

```
$ sudo service docker stop
$ sudo docker -d -H tcp://0.0.0.0:4243
```

`docker -d -H tcp://0.0.0.0:4243` 命令中，使用 `-d` 选项表示以守护进程模式运行，使用 `-H` 选项设置要接收连接的 IP 地址与端口号。Docker 守护进程的默认端口号为 4243。

运行命令时，Docker 守护进程以 foreground 方式运行。为了方便测试，运行新终端。

由于 Docker Remote API 采用 HTTP REST 格式，所以使用 `curl` 命令可以很方便地使用它们。首先下载 nginx 镜像。

```
$ curl -X POST http://127.0.0.1:4243/images/create?fromImage=nginx:latest
```

- ▶ `-X`：设置使用 POST 方法。
- ▶ 下载镜像的 Remote API 为 `/images/create?fromImage=< 镜像名 >:< 标签 >`。

使用下载的 nginx 镜像创建容器。

```
$ curl -X POST -H "Content-Type: application/json" \
-d '{ "Image": "nginx:latest", "ExposedPorts": {"80/tcp": {} }' \
http://127.0.0.1:4243/containers/create
{"Id": "386147f1c71ca7ee6a7013dbd2fff5d3091e41268885f954046be964b9ade39e", "Warnings": null}
```

> docker/book/Chapter14/docker-run.py

- -X: 设置使用 POST 方法。
- -H: 将 Content-Type 设置为 application/json。
- -d: 发送用于创建容器的设置数据。此处将 Image 设置为 nginx:latest, 将 ExposedPorts 设置为 80/tcp, 指定连接主机的端口。
- 创建容器的 Remote API 为 /containers/create。

容器创建完成后, 显示容器 ID。

下面启动创建的容器。

```
$ curl -X POST -H "Content-Type: application/json" \
  -d '{"PortBindings": {"80/tcp": [{"HostPort": "80"}]}}' \
  http://127.0.0.1:4243/containers/386147f1c71c/start
```

- -X: 设置使用 POST 方法。
- -H: 将 Content-Type 设置为 application/json。
- -d: 将容器的 80 号端口暴露在外。此处将 PortBindings 设置为 {"80/tcp": [{"HostPort": "80"}]}。
- 启动容器的 Remote API 为 /containers/<容器 ID>/start。容器 ID 使用前面创建容器时显示的值 (12 位)。

显示容器列表。

```
$ curl http://127.0.0.1:4243/containers/json
[{"Command": "nginx", "Created": 1409246048, "Id": "386147f1c71ca7ee6a7013dbd2fff5d3091e41268885f954046be964b9ade39e", "Image": "nginx:1", "Names": ["/angry_bartik"], "Ports": [{"IP": "0.0.0.0", "PrivatePort": 80, "PublicPort": 80, "Type": "tcp"}], "Status": "Up 6 seconds"}]
```

显示容器列表的 Remote API 是 GET 方法中使用的 /containers/json。若想将停止的容器也一并显示, 只要在后面添加 all=1 即可, 即 /containers/json?all=1。

像这样, 使用 HTTP 方法就可以控制 Docker 守护进程。事实上, 实际应用中很少只使用纯 HTTP REST 格式的 Remote API。此外, 由于 Remote API 升级速度很快, 本书无法涵盖所有 API。关于 Remote API 的更多介绍, 请参考如下 URL。

- https://docs.docker.com/reference/api/docker_remote_api/

13.1 ▶ 使用 Docker Remote API Python 库

前面通过 HTTP 方法使用了 Remote API。虽然可以使用纯 HTTP REST 格式的 Remote API，但已经出现许多针对不同语言的 Remote API 客户端库。本书将介绍 Docker 正式发布的 Python 库。

- ▶ 各语言版本库：https://docs.docker.com/reference/api/remote_api_client_libraries
- ▶ Python 库：<https://github.com/docker/docker-py>

请从我的 GitHub 仓库下载示例文件。

- ▶ <https://github.com/pyrasis/dockerbook>

使用 pip 可以安装 Docker Remote API Python 库。首先安装 pip 与 Python 开发包。

> Ubuntu

```
$ sudo apt-get install python-pip python-dev
```

> CentOS 6

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ sudo yum install python-pip python-devel
```

> CentOS 7

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
$ sudo yum install python-pip python-devel
```

提示 CentOS 7 EPEL 包版本

CentOS 7 EPEL 包版本升级速度快。若无法下载 rpm 文件，请先访问 http://dl.fedoraproject.org/pub/epel/7/x86_64/e/ 检查有无新版本，然后使用 yum 命令安装相应版本。

下面使用 pip 安装 docker-py 包。

```
$ sudo pip install docker-py
```

13.1.1 创建并启动容器

首先将如下内容保存为 docker-run.py。本示例先下载 nginx 镜像，再以容器运行。

- ▶ [dockerbook/Chapter14/docker-run.py](#)

> docker-run.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.pull(repository='nginx', tag='latest')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data'],
    name='hello'
)
c.start(
    container_id,
    port_bindings={80: ('0.0.0.0', 80)},
    binds={'/data': {'bind': '/data', 'ro': False}}
)
```

- 创建 `docker.Client` 类。在 `base_url` 中设置 Docker 守护进程的 Unix 套接字路径。
- 使用 `docker.Client` 类创建的实例 `c` 运行 `pull` 函数。设置镜像名与标签。
- 使用 `c.create_container` 函数创建容器。设置镜像名、连接主机的端口、与主机相连的目录、容器名。
- 使用 `c.start` 函数启动容器。使用运行 `c.create_container` 函数得到的容器对象，设置容器端口号与暴露在外的端口号，以及与主机目录相连的容器目录。

下面运行 `docker-run.py` 文件。

```
$ sudo python docker-run.py
```

显示容器列表后，使用 Python 库创建的容器也得到显示。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fda50393ec9f	nginx:1	"nginx"	1 sec...	Up 1 sec...	0.0.0.0:80->80/tcp	hello

`docker.Client` 类：创建运行 Docker 命令的默认对象。

- `base_url`：Docker 守护进程地址与端口号。
 - » Unix 套接字：`unix://`。
 - » 普通 HTTP 协议：`tcp://`，端口号 4243。
 - » 使用验证的 HTTPS 协议 (TLS)：`https://`，端口号 2376。
- `version`：使用指定版本的 Docker Remote API。与 Docker 运行文件的版本不同。

https://docs.docker.com/reference/api/docker_remote_api/

› timeout: 设置连接超时, 单位为秒。

```
c = docker.Client(base_url='unix://var/run/docker.sock',
                  version='1.12',
                  timeout=10)
```

pull 函数: 从仓库下载镜像。

- › repository: 要下载的镜像名, 格式为 <Docker Hub 用户账户>/<镜像名> 或 <镜像名>。
- › tag: 镜像标签。
- › stream: 接收结果值时, 使用 HTTP 1.1 Chunked transfer encoding。

```
c.pull(repository, tag=None, stream=False)
```

create_container 函数: 创建容器。

- › image: 要生成容器的镜像名, 格式为 <Docker Hub 用户账户>/<镜像名> 或 <镜像名>。
- › command: 容器启动时要运行的命令。例) '/bin/bash' 或 ['node', 'app.js']。
- › hostname: 容器主机名。
- › user: 以容器中的特定用户运行 command、entrypoint 中设置的命令。可以设置用户名或 UID。
- › detach: 以守护进程模式运行。docker run 命令的 -d 选项。
- › stdin_open: 连接基本输入 (stdin)。docker run 命令的 -i 选项。
- › tty: 分配 pseudo tty。docker run 命令的 -t 选项。
- › mem_limit: 设置内存限制。例) '1000000b'、'1000k'、'128m'、'1g'。
- › ports: 连接主机的端口。
 - › 例) [80, 443]
 - › 例) [(100, 'udp'), 200]
- › environment: 环境变量。例) ['Hello=1', 'Workd=2'] 或 {'Hello': '1', 'World': '2'}。
- › volumes: 要与主机连接的目录 (数据卷)。例) ['/data', '/www']。
- › network_disabled: 关闭网络。
- › name: 容器名。
- › entrypoint: 容器启动时运行的命令。关于 command 与 entrypoint 的不同, 请参考 7.6 节。
 - › 例) '/bin/bash' 或者 ['node', 'app.js']。

- `cpu_shares`: 分配 CPU 资源时的权重设置, 数字格式。
- `working_dir`: 要运行 `command`、`entrypoint` 设置命令的目录。
- `memswap_limit`: 设置内存换出限制, 数字格式。

```
c.create_container(image, command=None, hostname=None, user=None,
                  detach=False, stdin_open=False, tty=False, mem_limit=0,
                  ports=None, environment=None, dns=None, volumes=None,
                  network_disabled=False, name=None, entrypoint=None,
                  cpu_shares=None, working_dir=None, memswap_limit=0)
```

`start` 函数: 启动容器。

- `container`: 要启动的容器对象或名称。
- `binds`: 将容器目录绑定到主机的特定目录。`create_container` 函数中必须设置 `volumes`。`ro` 是读取专用选项。
 - » 例) `{ '/data': {'bind': '/data', 'ro': False}, '/www': {'bind': '/www', 'ro': False} }`
- `port_bindings`: 将要连接到主机的端口暴露在外。
 - » 例) `{ 80: ('0.0.0.0', 80), 443: ('0.0.0.0', 443) }`
 - » 例) `{ 80: ('0.0.0.0',), 443: ('0.0.0.0',) }`
 - » 例) `{ '100/udp': 100, 200: None }`
 - » 例) `{ 80: 80, 443: None }`
- `lxc_conf`: 使用 LXC 驱动程序的设置选项。
- `publish_all_ports`: 将连接主机的所有端口暴露在外。`docker run` 命令的 `-p` 选项。
- `links`: 连接容器的选项。例) `{ 'db': 'db', 'hello': 'hello' }`。
- `privileged`: 设置主机内核功能全部可用。
- `dns`: 添加 DNS 服务器设置。例)。
- `dns_search`: 添加搜索域名设置。例) `['168.126.63.1', '192.168.0.100']`。
- `volumes_from`: 数据卷容器设置。例) `['hello-volume', 'world-volume']`。
- `network_mode`: 网络模式设置。例) `'bridge', 'none', 'host'`。
- `restart_policy`: 设置容器终止时是否自动重启。
 - » 终止时总是重启: `{ 'MaximumRetryCount': 0, 'Name': 'always' }`
 - » 终止时重试 10 次: `{ 'MaximumRetryCount': 10, 'Name': 'on-failure' }`

```
c.start(container, binds=None, port_bindings=None, lxc_conf=None,
       publish_all_ports=False, links=None, privileged=False, dns=None,
       dns_search=None, volumes_from=None, network_mode=None, restart_policy=None)
```

13.1.2 创建镜像

首先将如下内容保存为 `docker-run.py` 文件。本示例中，先下载 `nginx` 镜像，再以容器运行。

➤ `dockerbook/Chapter14/docker-build.py`

> `docker-build.py`

```
import docker
import io

script = io.StringIO(u'\n'.join([
    'FROM ubuntu:14.04',
    'MAINTAINER Foo Bar <foo@bar.com>',
    'RUN apt-get update',
    'RUN apt-get install -y nginx',
    'RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf',
    'RUN chown -R www-data:www-data /var/lib/nginx',
    'VOLUME ["/data", "/etc/nginx/site-enabled", "/var/log/nginx"]',
    'WORKDIR /etc/nginx',
    'CMD ["nginx"]',
    'EXPOSE 80',
    'EXPOSE 443'
]))
```

```
c = docker.Client(base_url='unix://var/run/docker.sock')
c.build(tag='hello:0.1', quiet=True, fileobj=script, rm=True)
```

- 使用 `io.StringIO` 函数，生成 `Dockerfile` 的内容。
- 创建 `docker.Client` 类。在 `base_url` 中设置 Docker 守护进程的 Unix 套接字路径。
- 使用 `docker.Client` 类生成的实例 `c` 运行 `build` 函数。设置镜像名与标签。设置 `quiet=True` 防止创建镜像时显示输出结果，设置 `rm=True` 在镜像创建完成后删除临时镜像。将 `fileobj` 设置为前面创建的 `script`。

下面运行 `docker-build.py` 文件。

```
$ sudo python docker-build.py
```

显示镜像列表，可以看到 Python 库创建的镜像。

```
$ sudo docker ps
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
hello	0.1	8da15999194f	About an hour ago	263.8 MB
ubuntu	14.04	c4ff7513909d	2 weeks ago	225.4 MB

`build` 函数：创建镜像。

- path: Dockerfile 路径。不仅可以使用目录路径, 还可以使用 `http://`、`https://`、`git://`。
- tag: 镜像名与标签。
- quiet: 创建镜像时, 不显示输出结果。
- fileobj: 用对象形式设置 Docker 文件。若设置了 `fileobj`, 则忽略 `path`。
- nocache: 不缓存构建结果。
- rm: 镜像创建完成后, 删除临时镜像。
- stream: 接收结果值时, 使用 HTTP 1.1 Chunked transfer encoding。
- timeout: 设置连接超时, 以秒为单位。
- custom_context: 读取并使用 `tar`、`tar.gz` 等文件时, 设置为 `True`。文件内容必须设置在 `fileobj` 中。
- encoding: 若压缩 `custom_context`, 则该参数用于设置压缩格式。例) `'gzip'`。

```
c.build(path=None, tag=None, quiet=False, fileobj=None, nocache=False,
        rm=False, stream=False, timeout=None,
        custom_context=False, encoding=None)
```

下列示例中, 若当前目录存在 Dockerfile 文件, 则创建镜像。

> docker-build-local.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.build(path='.', tag='hello:0.1', quiet=True, rm=True)
```

如下示例中, 将 Dockerfile 及所需文件压缩为 `hello.tar.gz` 文件时, 创建镜像。

> docker-build-gzip.py

```
import docker

script = open('./hello.tar.gz', 'r')

c = docker.Client(base_url='unix://var/run/docker.sock')
c.build(tag='hello:0.1', quiet=True, fileobj=script,
        rm=True, custom_context=True, encoding='gzip')
```

下列示例中, 先从 GitHub 或网页下载 Dockerfile 文件, 再创建镜像。然而, 使用 `http://` 下载文件时, 将无法使用 `tar`、`tar.gz` 文件。

> docker-build-remote.py

```
import docker

c = docker.Client(base_url='unix:///var/run/docker.sock')
c.build(
    path='github.com/pyrasis/dind.git',      # GitHub
    #path='http://example.com/Dockerfile',    # http
    tag='hello:0.1', quiet=True, rm=True
)
```

13.1.3 显示容器列表

首先将如下内容保存为 docker-ps.py 文件。该示例用于显示容器列表。

> dockerbook/Chapter14/docker-ps.py

> docker-ps.py

```
import docker

c = docker.Client(base_url='unix:///var/run/docker.sock')
print c.containers(all=True)
```

- > 创建 docker.Client 类。将 base_url 设置为 Docker 守护进程 Unix 套接字路径。
- > 使用 docker.Client 类创建的实例 c 运行 containers 函数。设置 all=True 显示所有容器列表。

运行 docker-ps.py 文件，以 JSON 格式输出容器列表，如下所示。

```
$ sudo python docker-ps.py
```

```
[{'u'Status': 'u'Up 11 seconds', 'u'Created': 1409387349, 'u'Image': 'u'nginx:latest', 'u'Ports': [{'u'IP': 'u'0.0.0.0', 'u'Type': 'u'tcp', 'u'PublicPort': 80, 'u'PrivatePort': 80}], 'u'Command': 'u'nginx', 'u'Names': [u'/hello'], 'u'Id': 'u'6c70cda2b562c764c8d3cb605bdd2734fe32baf5c5ca6a7076180bf49297566e'}]
```

containers 函数：显示容器列表。

- > quiet：只显示容器 ID。
- > all：显示所有容器，包括停止的容器。
- > trunc：只显示部分输出结果。
- > latest：只显示标签为 latest 的容器。
- > since：显示指定容器之后创建的容器。
- > before：显示指定容器之前创建的容器。

➤ **limit**: 显示容器的最大个数。设置为 -1 表示显示所有容器。

```
c.containers(quiet=False, all=False, trunc=True, latest=False, since=None,
             before=None, limit=-1)
```

13.1.4 显示镜像列表

首先将如下内容保存为 `docker-images.py` 文件。该示例用于显示镜像列表。

➤ `dockerbook/Chapter14/docker-images.py`

> `docker-images.py`

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.images()
```

运行 `docker-images.py` 文件，以 JSON 格式输出镜像列表，如下所示。

```
$ sudo python docker-images.py
[{'Created': 1407814247, 'VirtualSize': 225406429, 'ParentId': 'u'cc58e55aa5a53b572f3b9009eb07e50989553b95a1545a27dceec830939892dba', 'RepoTags': ['u'ubuntu:14.04'], 'Id': 'u'c4ff7513909dedf4ddf3a450aea68cd817c42e698ebccf54755973576525c416', 'Size': 0}, {'Created': 1405934538, 'VirtualSize': 499135815, 'ParentId': 'u'5b9d57417804c431880c93e1adcc41afe6bf64513096803f03840117688f921a', 'RepoTags': ['u'nginx:latest'], 'Id': 'u'61e8f94e1d65cf3f2f409c70ecbc4401a9c5db83e9cfbf82c4e595e44a890376', 'Size': 0}]
```

`images` 函数：显示镜像列表。

➤ **name**: 只显示指定名称的镜像。

➤ **quiet**: 只显示镜像 ID。

➤ **all**: 显示所有镜像，包括临时镜像。

```
c.images(name=None, quiet=False, all=False)
```

13.1.5 其他示例与函数

`attach` 函数：连接容器。

➤ **container**: 要连接的容器对象或名称。

➤ **stdout**: 连接标准输出 (`stdout`)。

➤ **stderr**: 连接标准错误 (`stderr`)。

- › stream: 接收结果时, 使用 HTTP 1.1 Chunked transfer encoding。
- › logs: 显示日志。

```
c.attach(container, stdout=True, stderr=True, stream=False, logs=False)
```

若想使用 Python 实现输入输出, 只要使用 dockerpty 即可。

- › <https://github.com/d11wtq/dockerpty>

```
~$ git clone https://github.com/d11wtq/dockerpty.git
~$ cd dockerpty
~/dockerpty$ sudo python setup.py install
```

下列示例使用 dockerpty 在终端进行输入输出。

> docker-attach.py

```
import docker
import dockerpty

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='ubuntu:14.04',
    stdin_open=True,
    tty=True,
    command='/bin/bash'
)

dockerpty.start(c, container_id)
```

commit 函数: 将容器保存为镜像。

- › container: 要使用提交功能的容器对象或名称。
- › repository: 要创建的镜像名称, 格式为 '<Docker Hub 用户账户>/<镜像名>' 或 '<镜像名>'。
- › tag: 要创建的镜像标签。
- › message: 提交日志信息。
- › author: 创建镜像的作者信息。例) 'Hong, Gidong ,gd@yuldo.com'。
- › conf: 创建镜像时需要的设置值。

```
c.commit(container, repository=None, tag=None, message=None, author=None,
conf=None)
```

> docker-commit.py

```

import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.pull(repository='nginx', tag='latest')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data'])
)
c.commit(
    container_id,
    repository='hello',
    tag='0.1',
    message='example message',
    author='Hong, Gildong <gd@yuldo.com>',
    conf={
        'Hostname': '',
        'User': '',
        'Memory': 0,
        'MemorySwap': 0,
        'AttachStdin': False,
        'AttachStdout': True,
        'AttachStderr': True,
        'PortSpecs': None,
        'Tty': False,
        'OpenStdin': False,
        'StdinOnce': False,
        'Env': None,
        'Cmd': [
            '/bin/bash'
        ],
        'Volumes': {
            '/data': {}
        },
        'WorkingDir': '',
        'DisableNetwork': False,
        'ExposedPorts': {
            '80/tcp': {}
        }
    }
)

```

copy 函数：从容器中获取文件。

➤ **container：**要获取文件的容器对象或名称。

➤ **resource：**容器内部文件路径。

➤ 返回值为 `urllib3.response.HTTPResponse`，`.data` 中存放着 tar 格式的数据。

```
c.copy(container, resource)
```

> docker-cp.py

```
import docker
import tarfile
import io

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
response = c.copy(container_id, '/etc/nginx.conf')
t = tarfile.open(fileobj=io.BytesIO(response.data))
t.extractall();
```

diff 函数：显示镜像与容器间更改的部分。

> container：要比较的容器对象或名称。

```
c.diff(container)
```

> docker-diff.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
print c.diff(container_id)
```

export 函数：将容器保存为 tar 文件。

> container：要保存为 tar 文件的容器对象或名称。

```
c.export(container)
```

> docker-export.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
response = c.export(container_id)
with open('./nginx.tar', 'wb') as f:
    f.write(response.data)
```

history 函数：显示镜像历史。

› image：要显示历史的镜像名称，格式为 '< 镜像名 >:< 标签 >' 或 '< 镜像名 >'。

```
c.history(image)
```

> docker-history.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.history('nginx:latest')
```

import_image 函数：使用 tar 文件创建镜像。

- › src：tar 文件的路径。
- › repository：要创建的镜像名称，格式为 '< Docker Hub 用户账户 >/< 镜像名 >' 或 '< 镜像名 >'。
- › tag：要创建的镜像标签。
- › image：获取已有的镜像。例) 'nginx:latest'。

```
c.import_image(src=None, repository=None, tag=None, image=None)
```

> docker-import.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.import_image(src='./nginx.tar', repository='hello', tag='0.1')
```

info 函数：显示当前 Docker 的容器、镜像个数、Execution Driver、Storage Driver、内核版本等信息。

```
c.info()
```

> docker-info.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.info()
```

inspect_container 函数：显示容器的详细信息。

➤ **container：**要显示详细信息的容器对象或名称。

```
c.inspect_container(container)
```

> docker-inspect-container.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
print c.inspect_container(container_id)
```

inspect_image 函数：显示镜像的详细信息。

➤ **要显示详细信息的镜像 ID。**

```
c.inspect_image(image_id)
```

> docker-inspect-image.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.inspect_image('nginx:latest')
```

kill 函数：强制终止容器。

- container：要停止的容器对象或名称。
- signal：向容器发送特定信号。例）'KILL'。

```
c.kill(container, signal=None)
```

> docker-kill.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
c.kill(container_id)
```

login 函数：登录 Docker Hub。

- username：Docker Hub 账户名。
- password：Docker Hub 账户密码。
- email：加入 Docker Hub 时使用的电子邮件地址。
- registry：注册表服务器地址。默认为 Docker Hub。

```
c.login(username, password=None, email=None, registry=None)
```

> docker-login.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.login(username='exampleuser',
        password='examplepassword',
        email='exampleuser@example.com')
```

logs 函数：容器的标准输出，输出标准错误。

- container：要输出日志的容器对象或名称。
- stdout：输出标准输出。
- stderr：输出标准错误。

› stream: 接收结果值时, 使用 HTTP 1.1 Chunked transfer encoding。

› timestamps: 显示时间戳。

```
c.logs(container, stdout=True, stderr=True, stream=False, timestamps=False)
```

> docker-logs.py

```
import docker
import time

c = docker.Client(base_url='unix://var/run/docker.sock')
c.pull('ubuntu', tag='14.04')
container_id = c.create_container(
    image='ubuntu:14.04',
    command='/bin/bash -c "while sleep 1; do echo 1; done"',
)
c.start(container_id)
time.sleep(5)
print c.logs(container_id)
```

port 函数: 访问从容器连接到主机并暴露在外的端口。

› container: 要访问的容器对象或名称。

› private_port: 要访问的端口号。例) 80。

```
c.port(container, private_port)
```

> docker-port.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(
    container_id,
    port_bindings={80: ('0.0.0.0', 80)}
)
print c.port(container_id, 80)
```

push 函数: 将镜像上传到仓库。

- repository: 镜像名, 格式为 '<Docker Hub 用户账户>/< 镜像名 >'。
- tag: 镜像标签。
- stream: 接收结果值时, 使用 HTTP 1.1 Chunked transfer encoding。

```
c.push(repository, tag=None, stream=False)
```

> docker-push.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
#c.push(repository='localhost:5000/hello', tag='0.1') # 私有仓库

c.login(username='exampleuser',
        password='examplepassword',
        email='exampleuser@example.com')
c.push(repository='exampleuser/hello', tag='0.1') # Docker Hub
```

remove_container 函数: 删除容器。

- container: 要删除的容器对象或名称。
- v: 将与容器相连的数据卷一起删除。
- link: 使用 docker run 命令的 --link 选项连接容器时, 只删除连接状态。例) 'web/db'。

```
c.remove_container(container, v=False, link=False)
```

> docker-remove-container.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.remove_container('hello-nginx')
```

remove_image 函数: 删除镜像。

- image: 要删除的镜像名。例) 'nginx:latest'。

```
c.remove_image(image)
```

```
import docker
```

```
c = docker.Client(base_url='unix://var/run/docker.sock')
c.remove_image('nginx:latest')
```

restart 函数：重启容器。

- **container：**要重启的容器对象或名称。
- **timeout：**指定到容器终止时的等待时间，单位为秒。

```
c.restart(container, timeout=10)
```

> docker-restart.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.restart('hello-nginx')
```

search 函数：在 Docker Hub 中搜索镜像。

- **term：**搜索关键字。

```
c.search(term)
```

> docker-search.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.search('nginx')
```

stop 函数：停止容器。

- **container：**要停止的容器对象或名称。
- **timeout：**指定到容器终止时的等待时间，单位为秒。

```
c.stop(container, timeout=10)
```

> docker-stop.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
c.stop('hello-nginx')
```

tag 函数：为镜像设置标签。

➤ image：镜像名。例）'hello:0.1'。

➤ repository：注册服务器的地址与镜像名。例）'192.168.0.10:5000/hello'。

➤ tag：要创建的镜像标签。

➤ force：强制设置标签。

```
c.tag(image, repository, tag=None, force=False)
```

> docker-tag.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
#c.tag('hello:0.1', 'localhost:5000/hello', '0.1') # 私人仓库
c.tag('hello:0.1', 'exampleuser/hello', '0.1') # Docker Hub
```

top 函数：显示容器中当前运行的进程列表。

➤ container：要显示进程列表的容器对象或名称。

```
c.top(container)
```

> docker-top.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.top('hello-nginx')
```

version 函数：显示 Docker 版本信息。

```
c.version()
```

> docker-version.py

```
import docker

c = docker.Client(base_url='unix://var/run/docker.sock')
print c.version()
```

wait 函数：一直等到容器停止。容器终止时，显示 Exit Code。

➤ container: 要待机的容器对象或名称。

```
c.wait(container)
```

> docker-wait.py

```
import docker
```

```
c = docker.Client(base_url='unix://var/run/docker.sock')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data']
)
c.start(container_id)
print c.wait(container_id)
```

13.2 使用 Docker Remote API Python 库进行 HTTPS 通信

Docker 守护进程默认不提供使用 ID 与密码进行登录的功能，所以提供了利用证书进行 HTTPS 通信的功能。客户端只有拥有证书，才能与服务器端进行通信。

13.2.1 创建证书

使用 OpenSSL，创建服务器证书与客户端证书。

首先编辑 /etc/hosts 文件，添加测试域名。必须拥有 root 权限才可修改该文件。

> /etc/hosts

```
127.0.0.1    localhost
127.0.1.1    ubuntu
127.0.0.1    docker.example.com

# The following lines are desirable for IPv6 capable hosts
::1         localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
```

将 127.0.0.1 设置为 docker.example.com。事实上，也可以使用购买的域名，即使未购买也可使用。本书示例讲解以 docker.example.com 为例。

下面创建证书文件。以下命令整理于 dockerbook/Chapter14/cert.sh 文件。各位可以直接运行

cert.sh 文件，但必须亲自输入密码。

创建 CA (Certificate Authority) 证书文件。出现密码设置时，输入要使用的密码。创建 ca.pem 文件时，要输入 ca-key.pem 文件中设置的密码。

- Country Name: 国家代码。输入 CN。
- State or Province Name: 州或省名。请根据自身情况输入。
- Locality Name: 城市名。请根据自身情况输入。
- Organization Name: 输入公司名。
- Organization Unit Name: 输入组织名。
- Common Name: 运行 Docker 守护进程的服务器域名。若输入不正确，则即使创建证书也无法正常连接。根据 /etc/hosts 文件中的设置，输入 docker.example.com。
- Email Address: 电子邮件地址。

```
$ echo 01 > ca.srl
$ openssl genrsa -des3 -out ca-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:<输入要使用的密码>
Verifying - Enter pass phrase for ca-key.pem:<输入要使用的密码>
$ openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
Enter pass phrase for ca-key.pem:<输入ca-key.pem的密码>
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:KO
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:Seoul
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Company
Organizational Unit Name (eg, section) []:Example Company
Common Name (e.g. server FQDN or YOUR name) []:docker.example.com
Email Address []:exampleuser@example.com
```

创建服务器密钥、证书签名请求 (Certificate signing request) 文件。出现密码设置时，输入要使用的密码。创建 server.csr 文件时，要输入 server-key.pem 文件中设置的密码。

- '/CN=docker.example.com' 中一定要输入 Common Name 中设置的域名。

```
$ openssl genrsa -des3 -out server-key.pem 2048
```

```

Generating RSA private key, 2048 bit long modulus
.....++++
e is 65537 (0x10001)
Enter pass phrase for server-key.pem:<输入要使用的密码>
Verifying - Enter pass phrase for server-key.pem:<输入要使用的密码>
$ openssl req -subj '/CN=docker.example.com' -new -key server-key.pem \
-out server.csr
Enter pass phrase for server-key.pem:<输入server-key.pem的密码>

```

创建服务器证书文件。输入密码时，要输入 ca-key.pem 文件中设置的密码。

```

$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem \
-out server-cert.pem
Signature ok
subject=/CN=docker.example.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:<输入ca-key.pem的密码>

```

创建客户端密钥、证书签名请求文件。出现密码设置时，输入要使用的密码。创建 client.csr 文件时，要输入 key.pem 文件中设置的密码。

```

$ openssl genrsa -des3 -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....++++
e is 65537 (0x10001)
Enter pass phrase for key.pem:<输入要使用的密码>
Verifying - Enter pass phrase for key.pem:<输入要使用的密码>
$ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
Enter pass phrase for key.pem:<输入key.pem的密码>

```

创建设置文件，以允许使用证书文件进行连接。创建客户端证书文件。在密码输入部分输入 ca-key.pem 文件中设置的密码。

```

$ echo extendedKeyUsage = clientAuth > extfile.cnf
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \
-out cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:<输入ca-key.pem的密码>

```

删除服务器端使用的 server-key.pem 文件以及客户端使用的 key.pem 文件的密码。若出现密码输入请求，则输入之前设置的密码。


```
$ openssl rsa -in server-key.pem -out server-key.pem
Enter pass phrase for server-key.pem:<输入server-key.pem的密码>
writing RSA key
$ openssl rsa -in key.pem -out key.pem
Enter pass phrase for key.pem:<输入key.pem的密码>
writing RSA key
```

至此，证书创建完成。下面使用证书运行 Docker 守护进程。

```
$ sudo service docker stop
$ sudo docker -d --tlsverify --tlscacert=ca.pem --tlscert=server-cert.pem \
--tlskey=server-key.pem -H tcp://0.0.0.0:2376
```

- -d: 通过守护进程运行 Docker。
- --tlsverify: 用证书控制连接。
- --tlscacert: 设置前面创建的 ca.pem 文件。
- --tlscert: 设置前面创建的 server-cert.pem 文件。
- --tlskey: 设置前面创建的 server-key.pem 文件。
- -H: 接收连接的 IP 地址与端口号。设置为 tcp://0.0.0.0:2376。使用证书的守护进程的默认端口号为 2376。

执行命令，以 foreground 运行 Docker 守护进程。为了进行测试，打开新终端。

在当前 Linux 账户的 home 目录（/home/<用户账户>）创建 .docker 目录，并将 ca.pem、cert.pem、key.pem 文件复制到其中。

```
$ mkdir ~/.docker
$ cp ca.pem ~/.docker/ca.pem
$ cp cert.pem ~/.docker/cert.pem
$ cp key.pem ~/.docker/key.pem
```

若想从另一台电脑连接 Docker 守护进程，只要复制 .docker 目录使用即可。

下面连接 Docker 守护进程，并运行命令。

```
$ sudo docker --tlsverify -H docker.example.com:2376 info
Containers: 1
Images: 3
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Dirs: 5
Execution Driver: native-0.2
Kernel Version: 3.13.0-24-generic
Operating System: Ubuntu 14.04 LTS
WARNING: No swap limit support
```

输入的命令格式为 `docker --tlsverify -H <Docker 守护进程域名>:2376`。在 `-H` 选项中设置前面创建证书时输入的域名。若该域名不正确，则无法运行 Docker 命令。此外，若 `.docker` 目录中没有证书文件或者证书不对，则也无法运行 Docker 命令。

13.2.2 使用 Python 库

下面使用 Python 库连接使用证书的 Docker 守护进程 (TLS)。

将下列内容保存为 `docker-run-tls.py` 文件。连接使用证书的 Docker 守护进程，下载 nginx 镜像，并以容器运行。

➤ `dockerbook/Chapter14/docker-run-tls.py`

➤ `docker-run-tls.py`

```
import docker
from os.path import expanduser
home = expanduser('~')

tls_config = docker.tls.TLSConfig(
    client_cert=(home + '/.docker/cert.pem', home + '/.docker/key.pem'),
    ca_cert=home + '/.docker/ca.pem',
    verify=True
)
c = docker.Client(base_url='https://docker.example.com:2376', tls=tls_config)
c.pull(repository='nginx', tag='latest')
container_id = c.create_container(
    image='nginx:latest',
    ports=[80],
    volumes=['/data'],
    name='hello'
)
c.start(
    container_id,
    port_bindings={80: ('0.0.0.0', 80)},
    binds={'/data': {'bind': '/data', 'ro': False}}
)
```

- 创建 `docker.tls.TLSConfig` 类。将 `client_cert` 设置为 `cert.pem`、`key.pem` 文件的路径，将 `ca_cert` 设置为 `ca.pem` 文件的路径。设置时一定要使用绝对路径，使用 `expanduser` 函数获取当前 Linux 用户的 `home` 目录后，设置 `.docker` 目录。设置 `verify=True`，以允许使用证书进行连接。
- 创建 `docker.Client` 类。将 `base_url` 设置为 `https` 协议创建证书时输入的域名，如 `https://docker.example.com:2376`。在 `tls` 中设置前面创建的 `tls_config`。
- 使用 `docker.Client` 类创建的实例 `c` 运行 `pull` 函数。设置镜像名与标签。

- ▶ 使用 `c.create_container` 函数创建容器。设置镜像名、连接主机的端口、与主机连接的目录、容器名。
- ▶ 使用 `c.start` 函数启动容器。使用 `c.create_container` 函数创建的容器对象，设置容器端口号与暴露在外的端口号，以及与主机目录相连的容器目录。

下面运行 `docker-run-tls.py` 文件。

```
$ sudo python docker-run-tls.py
```

显示容器列表，并显示使用 Python 库创建的容器。

```
$ sudo docker --tlsverify -H docker.example.com:2376 ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
18e2d04f8753	nginx:1	"nginx"	1 sec...	Up 1 sec...	0.0.0.0:80->80/tcp	hello

-H 选项中一定要设置为创建证书时输入的域名，如 `docker.example.com:2376`。

第 14 章

DOCKER

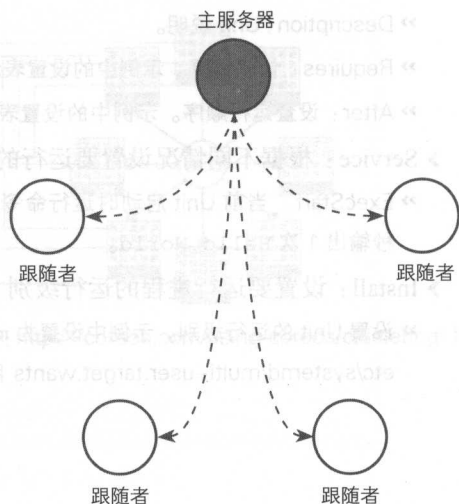
使用 CoreOS

CoreOS 是 Docker 专用 Linux 发布版本，它可以轻松将 Docker 容器部署到多台服务器，并提供集群、动态扩展、高可靠性（High Availability）等功能。

CoreOS 大致由 etcd、systemd、fleet 三种组件构成。首先，etcd 是一个应用于分布式环境的键值存储系统（Distributed Key-Value），用于存储并共享集群设置值与节点信息，类似于 Apache Zookeeper、doozer。

etcd 具有如下特征：

- ▶ 为 HTTP 协议提供 JSON 格式的 API。
- ▶ 提供每秒 1000 次的写性能。
- ▶ 利用 raft 一致性算法，从多台服务器中选出主服务器。
- ▶ 提供键自动删除功能（TTL，Time to live）。
- ▶ 提供 Atomic 读写操作，保障数据始终一致。
- ▶ 通过 HTTP 长轮询（long-polling）监视键的更改项。



etcd 以日志形式保存所有键的更改项，主服务器将日志复制给各跟随者并分享数据。不仅主服务器，在各节点中添加键或更改值时，所有节点都会体现。

图 14-1 复制日志（数据）到集群的各跟随者（出处：<https://coreos.com/using-coreos/etcd/>）

systemd 是 Linux 的服务管理器，用以取代已有的 System V、BSD init 系统。在 CoreOS 中通过 systemd 运行 Docker 容器。

- ▶ 与已有的 init 系统相比，启动速度更快。
- ▶ 可以设置服务之间的依赖关系，也可以控制运行顺序。
- ▶ 各守护进程日志可以使用 journald 轻松访问。

systemd 采用如下格式的 Unit 文件运行服务。只要在 `/etc/systemd/system` 目录下保存为 `example.service` 即可。

> `/etc/systemd/system/example.service`

```
[Unit]
Description=Example Service
Requires=docker.service
After=docker.service

[Service]
ExecStart=/usr/bin/docker run busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

▶ Unit：Unit 运行设置。

- » Description：Unit 说明。
- » Requires：依赖设置。示例中的设置表示，若要运行服务则需要 `docker.service`。
- » After：设置运行顺序。示例中的设置表示，`docker.service` 完成运行后再运行当前服务。

▶ Service：根据不同情况设置要运行的命令。

- » ExecStart：当前 Unit 启动时运行命令。示例中使用 `busybox` 镜像创建容器后，在 shell 中每隔 1 秒输出 1 次 `Hello World`。

▶ Install：设置要运行进程的运行级别（target）。

- » 设置 Unit 的运行级别。示例中设置为 `multi-user.target`，`example.service` 文件被链接到 `/etc/systemd/multi-user.target.wants` 目录。

提示 运行级别

Linux 系统中，运行级别（Run Level）定义操作系统的运行模式。

0. Halt：系统终止。将运行级别设置为 0，代表系统终止。（runlevel0.target、poweroff.target）
1. Single-user Mode：单用户模式，也称系统恢复模式。（runlevel1.target、rescue.target）
2. Multi-user Mode：不带网络连接的多用户模式。（runlevel2.target、multi-user.target）
3. Multi-user Mode with Networking：带网络连接的多用户模式。（runlevel3.target、multi-user.target）
4. User definable：用户自定义模式。（runlevel4.target、multi-user.target）
5. Multi-user graphical Mode：GUI 模式。（runlevel5.target、graphical.target）
6. Reboot：重启。（runlevel6.target、reboot.target）

以上介绍的运行级别以 Linux Standard Base（LSB）为基准，不同的 Linux 发行版本略有不同。CoreOS 中没有 GUI 模式，主要使用 multi-user.target。

fleet 是使用 etcd 与 systemd 的分布式服务运行（init）系统。systemd 是管理本地服务的系统，而利用 fleet 系统可以在多台服务器远程运行服务。

如图 14-2 所示，利用 fleet 系统将 6 个 API 服务器容器与 2 个负载均衡容器部署到 CoreOS 集群。fleet 系统会根据集群的实际情况自动选择主机，然后部署 Docker 容器。当然，也可以选择设置特定主机以部署容器。

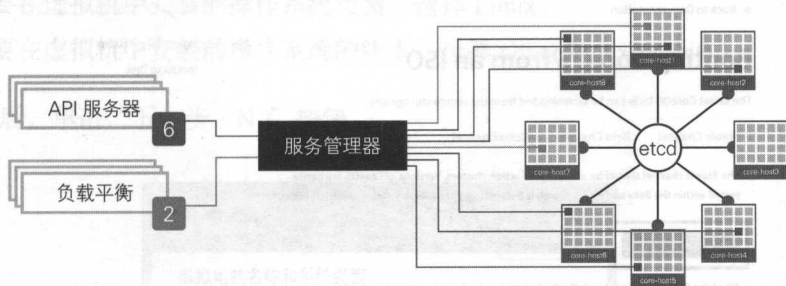


图 14-2 使用 fleet 向多台服务器部署 Docker 容器（出处：<https://coreos.com/using-coreos/clustering/>）

fleet 具有如下特征：

- 将 Docker 容器部署到任意主机。
- 将服务适当分散到多台主机，而不是蜂拥到特定主机。
- 即使某台主机发生故障，也将维持特定的服务个数。也就是说，某台主机停止时，其运行的服务会被转移到其他主机中运行。
- 自动发现集群中的主机。

请到我的 GitHub 仓库下载示例文件。

➤ <https://github.com/pyrasis/dockerbook>

14.1 在 VirtualBox 中安装 CoreOS

本节将在开源虚拟软件 VirtualBox (<https://www.virtualbox.org/>) 中安装 CoreOS。VirtualBox 的安装方法并无特别之处，不再另行说明。

首先访问 <https://coreos.com/docs/running-coreos/platforms/iso/>，然后单击 Download Stable ISO 按钮，下载 ISO 文件。

- Stable Channel：经过大量测试的稳定版本。
- Beta Channel：Alpha Channel 经过测试后注册的版本。
- Alpha Channel：最新版本。其中包含的 Docker、etcd、fleet 都是最新版本。

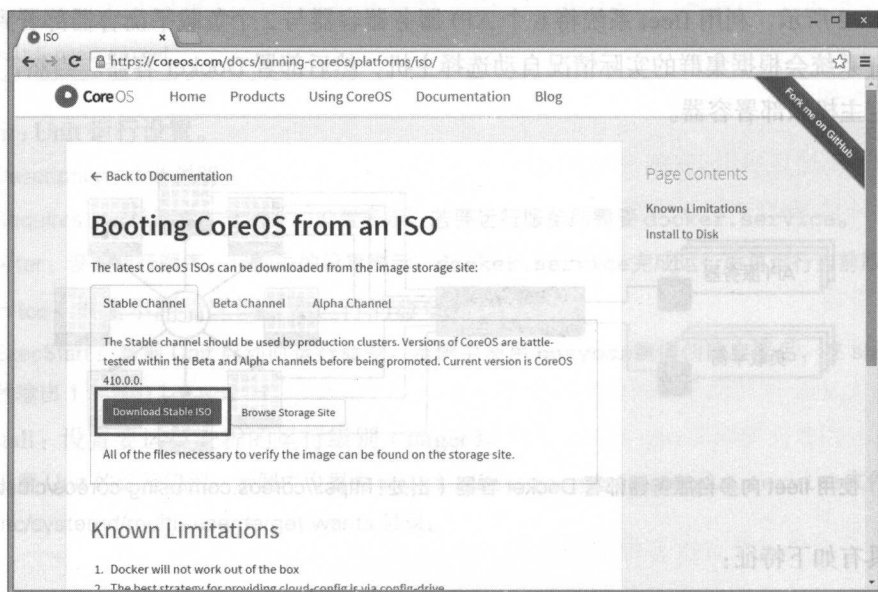


图 14-3 从 coreos.com 下载 ISO 文件

接着，运行 VirtualBox，单击“新建（N）”按钮。

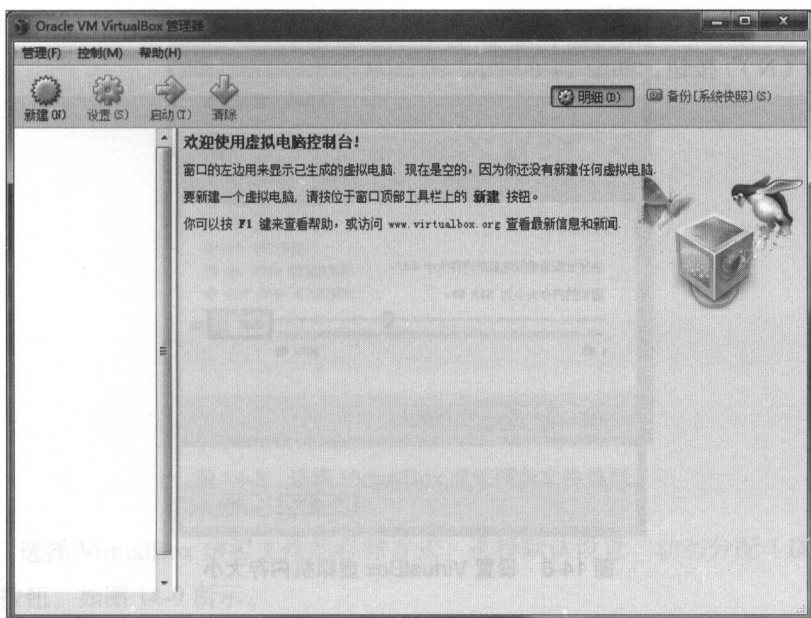


图 14-4 VirtualBox

创建 VirtualBox 虚拟机, 如图 14-5 所示。

- ▶ 名称: 虚拟机名称。输入 CoreOS。
- ▶ 类型: 要在虚拟机中安装的操作系统类型。选择 Linux。
- ▶ 版本: 要在虚拟机中安装的操作系统的版本。选择 Other Linux(64-bit)。

设置完成后, 单击“下一步(N)”按钮。

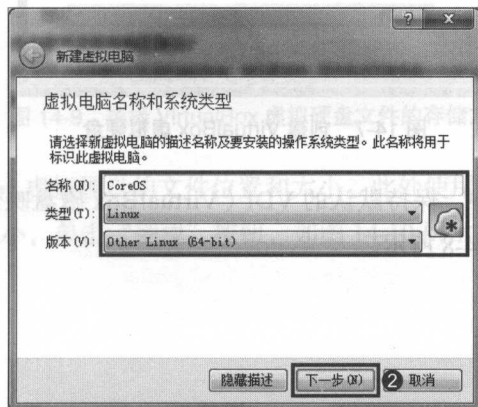


图 14-5 设置 VirtualBox 虚拟机名称与操作系统类型

设置虚拟机内存大小。我将内存设置为 2048 MB，请各位根据自身情况设置合适的大小。单击“下一步(N)”按钮，如图 14-6 所示。

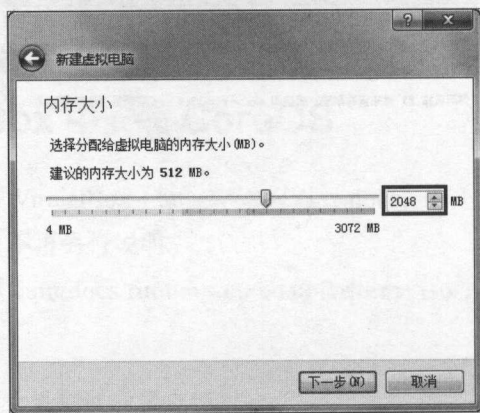


图 14-6 设置 VirtualBox 虚拟机内存大小

选择“现在创建虚拟硬盘(C)”，单击“创建”按钮，如图 14-7 所示。

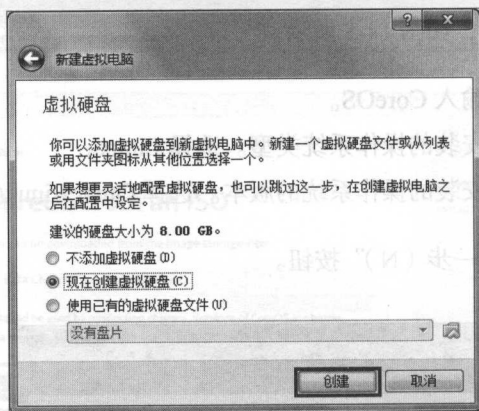


图 14-7 创建 VirtualBox 虚拟硬盘

“虚拟硬盘文件类型”中，保持默认的 VDI (VirtualBox 磁盘映像) 处于选中状态，单击“下一步(N)”按钮，如图 14-8 所示。

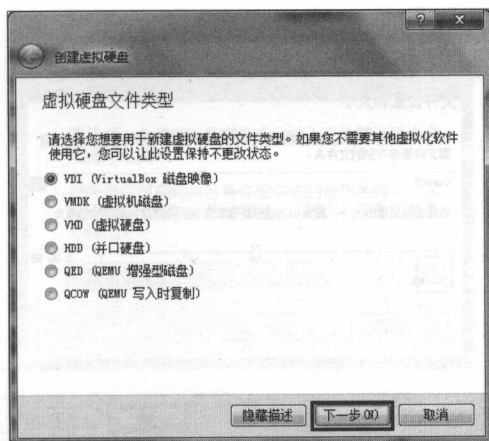


图 14-8 选择 VirtualBox 虚拟硬盘文件类型

接下来，选择 VirtualBox 虚拟文件的存储方式，选择默认设置“动态分配（D）”，单击“下一步（N）”按钮，如图 14-9 所示。

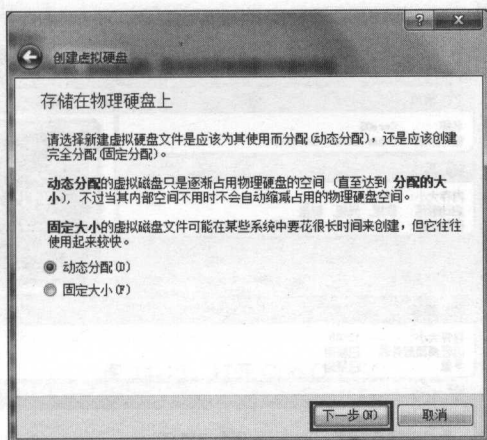


图 14-9 选择 VirtualBox 虚拟硬盘文件的存储方式

接着，设置 VirtualBox 虚拟硬盘的文件位置和大小。此处使用默认设置 8.00 GB，请各位根据自身情况设置合适的大小，单击“创建”按钮，如图 14-10 所示。

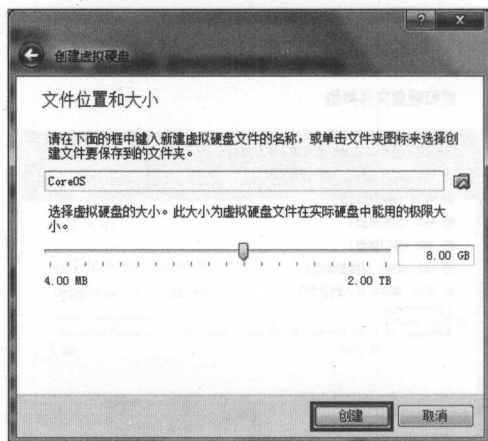


图 14-10 设置 VirtualBox 虚拟硬盘文件的位置和大小

至此, CoreOS 虚拟机创建完成。单击“启动 (T)”按钮。

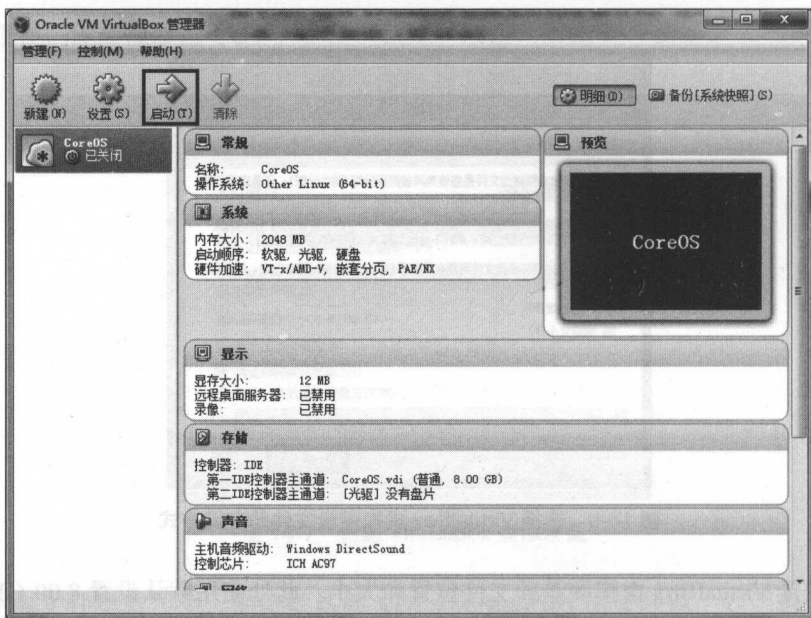


图 14-11 VirtualBox 虚拟机创建完成

启动虚拟机, 显示“选择启动盘”窗口, 如图 14-12 所示。单击右下角文件夹图标。选择前面下载的 `coreos_production_iso_image.iso` 文件, 单击“打开”按钮, 如图 14-13 所示。打开 CoreOS ISO 文件后, 单击“启动”按钮, 如图 14-14 所示。

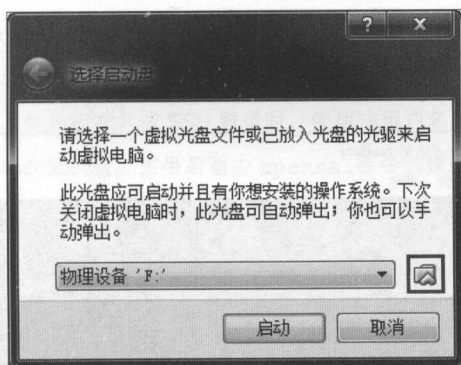


图 14-12 选择 VirtualBox 启动盘

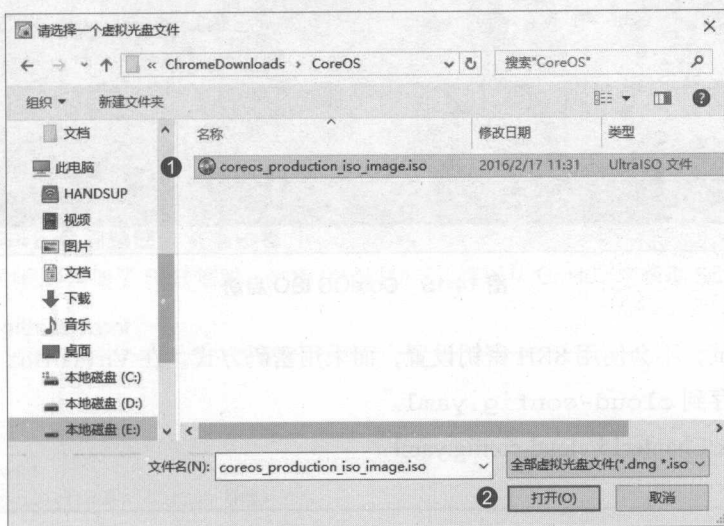


图 14-13 打开 CoreOS ISO

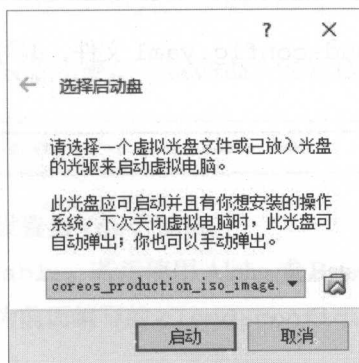


图 14-14 选择 VirtualBox 启动盘

从 CoreOS ISO(CD) 文件引导启动, 如图 14-15 所示。CoreOS 无另外的安装界面, 使用通过 ISO 文件启动的 `coreos-install` 命令可以将其安装到磁盘。

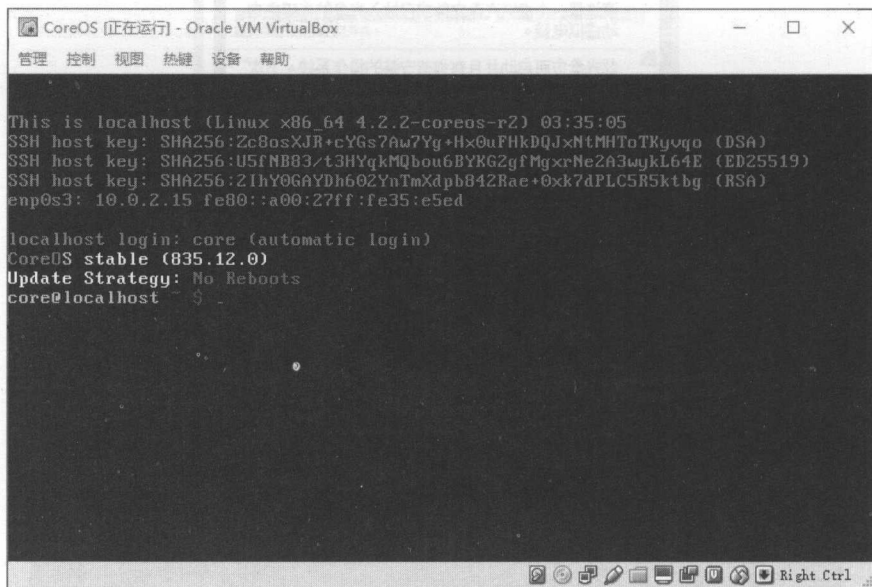


图 14-15 CoreOS ISO 启动

为了便于测试, 不会使用 SSH 密钥设置, 而采用密码方式。在 VirtualBox 中输入如下命令创建密码后, 保存到 `cloud-config.yaml`。

➤ `dockerbook/Chapter15/cloud-config.yaml`

```
$ openssl passwd -1 > cloud-config.yaml
Password: <输入要使用的密码>
Verifying - Password: <再次输入要使用的密码>
```

接下来, 使用 `vim` 打开 `cloud-config.yaml` 文件, 编写如下。

➤ `cloud-config.yaml`

```
#cloud-config

users:
- name: exampleuser
  passwd: $1$geiJttft8$vs8v2Rnlw1iNkeAFnPWQ00
  groups:
    - sudo
    - docker
```

➤ 必须在首行输入 `#cloud-config`。

➤ `users`：用户设置。

➤ `name`：用户账号名称。使用控制台或 SSH 登录时，使用该用户名。输入 `exampleuser`。

➤ `passwd`：用户账号密码的散列值。使用前面由 `openssl` 命令创建的密码散列值。

➤ `group`：设置用户账号组。设置为 `sudo` 组，以使用 `root` 权限运行命令。并且设置 `docker` 组，以在不使用 `sudo` 命令的情形下运行 `docker`。

提示 若要设置 SSH 密钥，编写如下。

> `cloud-config-ssh.yaml`

```
#cloud-config

users:
- name: exampleuser
  ssh-authorized-keys: ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQ<省略>
groups:
- sudo
- docker
```

若 GitHub 账号中已经注册了 SSH 密钥，则编写代码如下，可以从 GitHub 中获取 SSH 密钥。

> `cloud-config-github.yaml`

```
#cloud-config

users:
- name: exampleuser
  coreos-ssh-import-github: <GitHub 账号>
groups:
- sudo
- docker
```

运行如下命令，向 VirtualBox 虚拟硬盘（`/dev/sda`）安装 CoreOS。

```
$ sudo coreos-install -d /dev/sda -C stable -c cloud-config.yaml
```

➤ `-d`：待安装的硬盘设备。设置为 `/dev/sda`。

➤ `-C`：发布通道。设置为 `stable`。若想使用 Alpha 或 Beta 通道，请设置为 `alpha`、`beta`。

➤ `-c`：安装文件路径。设置为前面编写的 `cloud-config.yaml`。

短暂等待后，CoreOS 安装完成。输入如下信息，表示安装成功。

Success! CoreOS stable <版本> is installed on /dev/sda

在 VirtualBox 虚拟机画面中，依次单击“设备→分配光驱→移除虚拟盘”，如图 14-16 所示。



图 14-16 从 VirtualBox 中移除 CoreOS ISO 文件

弹出如图 14-17 所示窗口，单击“强制释放”按钮。

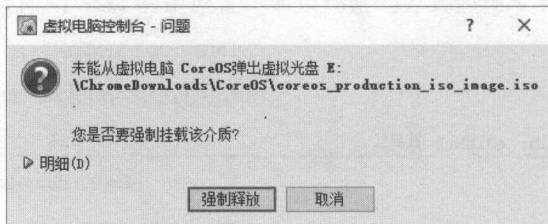


图 14-17 解除 VirtualBox 强制挂载

移除 ISO 文件后，输入如下命令，重启虚拟机。

```
$ sudo reboot
```

VirtualBox 虚拟机重启后，在 login 中输入 exampleuser、在 Password 中输入前面设置的密码，即可登录 CoreOS。当然，也可以不用 VirtualBox，而使用 SSH 进行连接。

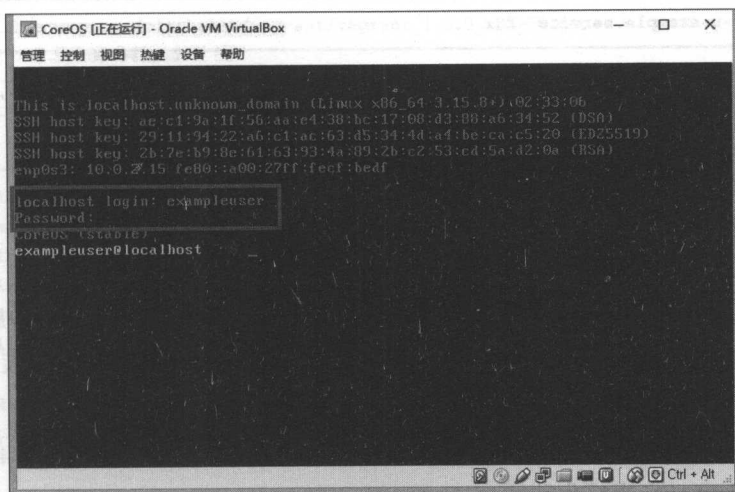


图 14-18 登录 CoreOS

使用 systemd 运行服务

首先，将如下内容保存到 `/etc/systemd/system` 目录的 `example.service` 文件。若想在 `/etc` 目录下创建文件，需要拥有 `root` 权限，所以要使用 `sudo` 命令。

➤ `dockerbook/Chapter15/example.service`

➤ `/etc/systemd/system/example.service`

```
[Unit]
Description=Example Service
Requires=docker.service
After=docker.service

[Service]
ExecStart=/usr/bin/docker run busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

接着，使用 `systemctl` 命令运行 `example.service`。

```
$ sudo systemctl start example.service
```

命令格式为 `systemctl start <system Unit 名称>`，若未输出任何错误，则表示服务正常运行。

使用 `journalctl` 命令输出 Docker 容器的日志。

```
$ sudo journalctl -u example.service -f
```

命令格式为 `journalctl -u <system Unit 名称>`。使用 `-f` 选项可以实时输出日志。

14.2 ▸ 使用 Vagrant 安装 CoreOS

前面学习了使用 VirtualBox 安装 CoreOS 的方法。由于 CoreOS 安装在一台虚拟机中，所以除了运行 CoreOS 之外没有太大意义。下面使用 Vagrant 创建 3 个虚拟机并安装 CoreOS，构建集群。

使用 Vagrant 工具可以轻松创建并管理虚拟机。但 Vagrant 并不是虚拟化软件，所以必须安装 VirtualBox 或 VMware 虚拟化软件。

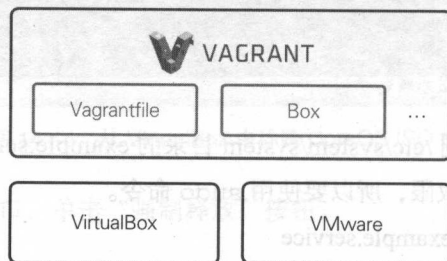


图 14-19 Vagrant 概念图

提示 Vagrant 与 Docker

与 Docker 类似，Vagrant 也通过 Vagrant Cloud (<https://vagrantcloud.com/>) 对外提供各种 Linux 发行版、Web 服务器、DB 等镜像 (Box)。

Vagrant 平台基于 VirtualBox 与 VMware 虚拟软件，而 Docker 平台则基于 Linux 内核的 cgroups、namespaces 的。与 Docker 镜像不同，Vagrant Box 具备完整的操作系统形态。

之前使用 VirtualBox 安装 CoreOS 时，需要在终端中逐个创建设置文件并运行安装命令。使用 Vagrant 可以通过事先设置好的环境自动创建虚拟机，并安装配置操作系统。

在 Windows 与 Mac OS X 中进入下列地址，下载安装文件安装即可。安装过程并无特别之处，故省略。

▸ Windows, Mac OS X: <https://www.vagrantup.com/downloads.html>

> Ubuntu

```
$ sudo apt-get install virtualbox
```



```
$ wget https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.deb
$ dpkg -i vagrant_1.6.5_x86_64.deb
```

> CentOS 6

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ curl http://download.virtualbox.org/virtualbox/rpm/el/virtualbox.repo -o virtualbox.repo
$ sudo mv virtualbox.repo /etc/yum.repos.d/
$ sudo yum install VirtualBox-4.3 kernel-devel dkms
$ sudo yum groupinstall "Development Tools"
$ sudo yum install https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.rpm
$ sudo su
# export KERN_DIR=/usr/src/kernels/2.6.32-431.23.3.el6.x86_64
# /etc/init.d/vboxdrv setup
```

> CentOS 7

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
$ curl http://download.virtualbox.org/virtualbox/rpm/el/virtualbox.repo -o virtualbox.repo
$ sudo mv virtualbox.repo /etc/yum.repos.d/
$ sudo yum install VirtualBox-4.3 kernel-devel dkms
$ sudo yum groupinstall "Development Tools"
$ sudo yum install https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.rpm
$ sudo su
# export KERN_DIR=/usr/src/kernels/3.10.0-123.6.3.el7.x86_64
# /etc/init.d/vboxdrv setup
```

提示 CentOS7 EPEL 包版本

CentOS 7 EPEL 包版本升级速度快。若无法下载 rpm 文件，请先访问 http://dl.fedoraproject.org/pub/epel/7/x86_64/e/ 检查有无新版本，然后使用 yum 命令安装相应版本。

接下来，使用 Git 下载已经编写好用于 CoreOS 的 Vagrantfile 文件。（现在开始出现的命令同时适用于 Mac OS X、Linux、Windows 系统，在 Windows 中运行命令行工具或 PowerShell；在 Mac OS X 中运行终端；在 Linux 中直接在终端中执行相应命令。）关于 Git 的安装方法，请参考 8.1.1 节。

```
git clone https://github.com/coreos/coreos-vagrant.git
cd coreos-vagrant
```

为了保存集群状态，必须分配获得 etcd discovery URL。在终端中运行如下命令，或者在 Web 浏览器中访问 <https://discovery.etcd.io/new>。


```
$ curl -w "%n" https://discovery.etcd.io/new
https://discovery.etcd.io/c0d96ba6f4024d5cabe2484e89bca52f
```

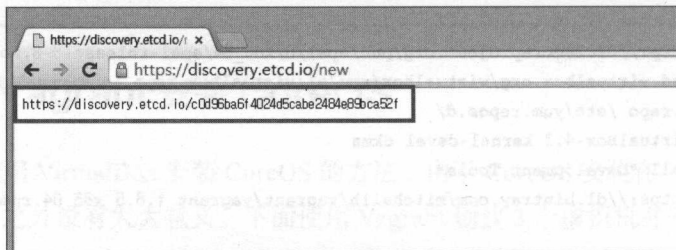


图 14-20 分配 etcd discovery URL

为便于使用 etcd，CoreOS 中以服务形式提供 discovery URL。该 URL 用于选出 etcd Leader Node (master)。

将如下内容保存到 coreos-vagrant 目录下的 user-data 文件。

➤ dockerbook/Chapter15/vagrant/user-data

> user-data

```
#cloud-config
```

```
coreos:
```

```
  etcd:
```

```
    discovery: https://discovery.etcd.io/<集群ID>
```

```
    addr: $public_ipv4:4001
```

```
    peer-addr: $public_ipv4:7001
```

```
  fleet:
```

```
    public-ip: $public_ipv4
```

```
  units:
```

```
    - name: etcd.service
```

```
      command: start
```

```
    - name: fleet.service
```

```
      command: start
```

➤ #cloud-config 必须在首行输入。

➤ coreos: CoreOS 的主设置。

➤ etcd: etcd 设置。

➤➤ discovery: discovery URL。设置为前面分配得到的 discovery URL (设置时一定要包含 https://discovery.etc.io 与集群 ID)。

➤➤ addr: clientURL IP 地址, 设置为当前服务器的 IP 地址。若使用 \$public_ipv4, 则 vagrant 自动为虚拟机设置分配到的 IP 地址。

➤➤ peer-addr: peerURL IP 地址, 设置为当前服务器的 IP 地址。同样, 若使用 \$public_ipv4, 则

vagrant 自动为虚拟机设置分配到的 IP 地址。

➤ **fleet: fleet 设置。**

» **public-ip:** 设置当前服务器的 IP 地址。若使用 `$public_ipv4`, 则 vagrant 自动为虚拟机设置分配到的 IP 地址。

➤ **units: 设置要运行的服务。**

» **name:** 服务名称。此处运行 `etcd.service.fleet.service`。

» **command:** 要向服务下达的命令。此处设置为 `start`, 表示启动服务。

提示 `$public_ipv4`、`$private_ipv4` 不仅可以用在 Vagrant 中, 还可以用在 Amazon EC2、Google Compute Engine、OpenStack、Rackspace、DigitalOcean 中, 用于自动设置当前服务器的公共 IP 地址以及私有 IP 地址。

提示 设置元数据

若要设置元数据, 只要向 fleet 添加 metadata 项目即可, 如下所示。使用 `fleetctl` 在元数据一致的节点中执行 Unit。

• `dockerbook/Chapter15/metadata/user-data`

coreos:

fleet:

public-ip: `$public_ipv4`

metadata: `region=ap-northeast-1,disk=ssd,platform=cloud,provider=amazon`

元数据无特定规则, 各位可根据自身习惯进行设置。各元数据以逗号 (,) 分隔。

将如下内容保存到 `coreos-vagrant` 目录下的 `config.rb` 文件。

➤ `dockerbook/Chapter15/config.rb`

> **config.rb**

```
$num_instances=3
$update_channel='stable'
```

将 `$num_instances` 设置为 3, 创建 3 台虚拟机。将 `$update_channel` 设置为 `stable`, 使用 `Stable` 版本。

提示 `config.rg.sample` 与 `discovery URL`

本示例中, 先分配 `discovery URL`, 然后直接设置到 `user-data` 文件。查看 `config.rb.sample` 文件, 其中包含的代码在分配到 `discovery URL` 后自动将其设置到 `user-data` 文件。

接下来，在 `coreos-vagrant` 目录下运行如下命令，创建虚拟机。（在 Windows 系统中不要使用 `sudo`。）

```
$ sudo vagrant up
```

短暂等待后，虚拟机创建完成。Windows 系统中若弹出 VirtualBox 用户账号控制窗口，单击“是 (Y)”按钮。接着，输入 `vagrant status` 命令。

```
$ sudo vagrant status
```

```
Current machine states:
```

```
core-01          running (virtualbox)
core-02          running (virtualbox)
core-03          running (virtualbox)
```

This environment represents multiple VMs. The VMs are all listed above with their current state. For more information about a specific VM, run `vagrant status NAME`.

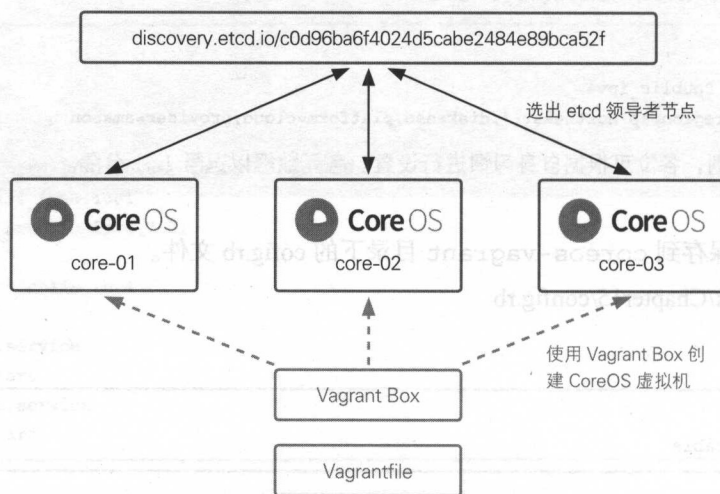


图 14-21 使用 vagrant 创建 CoreOS 虚拟机

显示安装有 CoreOS 系统的 `core-01`、`core-02`、`core-03` 虚拟机。使用 `vagrant ssh` 命令通过 SSH 连接 `core-01` 虚拟机。

```
$ sudo vagrant ssh core-01
```

```
Last login: Sun Sep 7 11:40:28 2014 from 10.0.2.2
```

```
CoreOS (stable)
```

```
core@core-01 ~ $
```

使用的命令格式为 `vagrant ssh <虚拟机名称>`，这样就轻松连接到了 `core-01` 虚拟机。

14.3 > 使用 etcd

我们前面使用 `vagrant` 构建了 3 台 CoreOS 虚拟机组成的集群，下面学习在该集群中使用 `etcd` 进行数据共享的方法。

运行 2 个终端（Windows 下的命令行工具、PowerShell、Mac OS X 的终端），转到 `coreos-vagrant` 目录，使用 `vagrant ssh` 命令连接到 CoreOS 虚拟机。

> 第一个终端

```
~$ cd coreos-vagrant
~/coreos-vagrant$ vagrant ssh core-01
```

> 第二个终端

```
~$ cd coreos-vagrant
~/coreos-vagrant$ vagrant ssh core-02
```

14.3.1 创建 etcd 键与目录

在 `core-01` 中运行如下命令，创建键并设置键值。

> core-01

```
$ etcdctl mk /hello world
```

命令格式为 `etcdctl mk <etcd 键路径> <值>`，`etcd` 路径以 `/` 开始。

接着，在 `core-02` 中运行如下命令，获取键值。

> core-02

```
$ etcdctl get /hello
world
```

命令格式为 `etcdctl get <etcd 键路径>`。输出前面设置的 `world` 值。

同样，在 `core-01` 中创建目录，在 `core-02` 中也能看到。

> core-01

```
$ etcdctl mkdir /hello-dir
```

命令格式 `etcdctl mk <etcd 目录路径>`。

14.3.2 输出 etcd 键与目录列表

etcd 拥有类似于普通文件系统的目录。运行如下命令，输出键与目录。

> core-01

```
$ etcdctl ls / --recursive
/coreos.com
/coreos.com/updateengine
/coreos.com/updateengine/rebootlock
/coreos.com/updateengine/rebootlock/semaphore
/hello
```

命令格式为 `etcdctl ls <etcd 路径>`，使用 `--recursive` 选项，将子目录与键全部输出。此处的 `/coreos.com` 目录是与 `CoreUpdate`（有偿服务）有关的设置。

etcd 设置值包含于 `_etcd` 目录，无法使用 `etcdctl ls` 命令查看。运行如下命令，查看 `_etcd` 目录中的内容。

> core-01

```
$ etcdctl ls /_etcd --recursive
/_etcd/machines
/_etcd/machines/b3318c7b2f63466e9f4065eachbda2b18
/_etcd/machines/d80aaff50718476ab5057c00804ee59b
/_etcd/machines/6960c4ce6f6b44518687a32d5e39dec9
/_etcd/config
```

由于创建了 3 台 CoreOS 虚拟机，故 `/_etcd/machines` 目录下存在 3 个键。

14.3.3 设置自动删除 etcd 键与目录

用户可以设置经过一定时间后将 etcd 键与目录自动删除。先使用 `etcdctl mk` 命令创建键，然后使用 `etcdctl ls` 命令输出 `/` 的目录。

> core-01

```
$ etcdctl mk /hello2 world --ttl 20
world
$ etcdctl ls /
/coreos.com
/hello2
/hello-dir
```

命令格式为 `etcdctl mk <etcd 路径>< 值> --ttl <秒>`。若创建键之后直接输出 / 的目录，则显示 /hello2 键。

经过 20 秒后，再输出 / 的目录。

> core-01

```
$ etcdctl ls /
/coreos.com
```

由于进行了自动删除设置，故删除 /hello2 键。同样，要删除目录，只要在 `etcdctl mkdir` 命令中使用 `--ttl` 选项即可。

> core-01

```
$ etcdctl mkdir /hello-dir2 --ttl 20
```

14.3.4 监视 etcd 键

使用 `etcdctl watch` 命令即可在添加 / 删除键或者修改键值时立即通知。

在 core-01 中创建键，然后运行 `etcdctl watch` 命令。

> core-01

```
$ etcdctl mk /hello3 world
world
$ etcdctl watch /hello3
```

命令格式为 `etcdctl watch <etcd 路径>`。

接着，在 core-02 中将 /hello3 的键值更改为 abcd1234。

> core-02

```
$ etcdctl set /hello3 abcd1234
abcd1234
```

命令格式为 `etcdctl set <etcd 路径>< 值>`。

在 core-01 中输入 `etcdctl watch` 命令后，即输出修改后的键值 abcd1234，如下所示。

> core-01

```
$ etcdctl watch /hello3
abcd1234
```

若使用 `etcdctl exec-watch` 命令，键发生变化时，可以运行特定命令。

> core-01

```
$ etcdctl exec-watch /hello3 -- sh -c "echo hello3 modified"
hello3 modified
```

命令格式为 `etcdctl exec-watch <etcd 路径> --sh -c "<命令>"`。此处若更改 `/hello3` 键，则通过 `sh` 运行 `echo hello3 modified`。

14.3.5 etcd 其他命令

删除键与目录的命令如下所示。

```
$ etcdctl rm /hello3
$ etcdctl rmdir /hello-dir
```

命令格式为 `etcdctl rm <etcd 键路径>`、`etcdctl rmdir <etcd 目录路径>`。
将键创建为目录的命令如下所示。

```
$ etcdctl setdir /hello3
```

命令格式为 `etcdctl setdir <etcd 键路径>`。

对已创建的键值或设置以及目录设置进行修改的命令如下所示。

```
$ etcdctl update /hello abcd1234 --ttl 20
$ etcdctl updatedir /hello-dir --ttl 20
```

- 命令格式为 `etcdctl update <etcd 键路径> <值> --ttl <秒>`。省略 `--ttl` 选项可以只更改键值。
- 命令格式为 `etcdctl updatedir <etcd 目录路径> --ttl <秒>`。不能省略 `--ttl` 选项。

14.4 > 使用 fleet

下面使用 `fleet` 在集群中运行服务。首先打开 3 个终端（Windows 的命令行、PowerShell、Mac OS X 的终端），转到 `coreos-vagrant` 目录后，使用 `vagrant ssh` 命令连接 CoreOS 虚拟机。

> 第一个终端

```
~$ cd coreos-vagrant
~/coreos-vagrant$ vagrant ssh core-01
```

> 第二个终端

```
~$ cd coreos-vagrant
~/coreos-vagrant$ vagrant ssh core-02
```

> 第三个终端

```
~$ cd coreos-vagrant
~/coreos-vagrant$ vagrant ssh core-03
```

14.4.1 输出 fleet 机器列表

运行如下命令，输出集群中机器（服务器、节点）的列表。

```
$ fleetctl list-machines
```

MACHINE	IP	METADATA
6960c4ce...	172.17.8.103	-
b3318c7b...	172.17.8.101	-
d80aaff5...	172.17.8.102	-

输出的信息分别为机器 ID、IP 地址、元数据。使用 `--full` 选项可以输出所有机器 ID。

14.4.2 使用 fleet 运行 Unit

在 core-01 中将如下内容保存为 `hello.service` 文件。fleet 运行的 Unit 文件（`.service`）也可以不在 `/etc/systemd/system` 目录而位于其他目录。

> `dockerbook/Chapter15/hello.service`

```
> ~/hello.service
```

```
[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello
ExecStartPre=/usr/bin/docker rm hello
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello
```

Unit: Unit 运行设置。

» Requires、After: 由于要使用 Docker, 故设置为 `docker.service`。

Service: 根据各自情况设置要运行的命令。

» ExecStartPre: 运行主命令前的准备工作。执行 `docker kill hello`、`docker rm hello` 命令, 若 `hello` 容器正在运行, 则停止后删除。像 `ExecStartPre=-` 一样, 在 `=` 之后使用 `-`, 即使发生错误也会直接跳过。不存在 `hello` 容器时, 若企图终止或删除, 将会引发错误, 故必须添加 `-` 符号。

» ExecStart: Unit 启动时要运行的主命令。使用 `busybox` 镜像创建 `hello` 容器, 每隔 1 秒输出 1 次 `HelloWorld`。

» ExecStop: Unit 终止时要运行的命令。运行 `docker stop hello` 命令停止容器。

在 `hello.service` 文件所在的目录中运行如下命令。

> core-01

```
$ fleetctl start hello.service
```

```
Job hello.service launched on d80aaff5.../172.17.8.102
```

命令格式为 `fleetctl start <Unit 文件>`。在集群的 `core-01`、`core-02`、`core-03` 任意虚拟机中运行 Unit 文件。我在 `core-02` (`d80aaff5`, `172.17.8.102`) 中运行 `hello.service` Unit 文件。

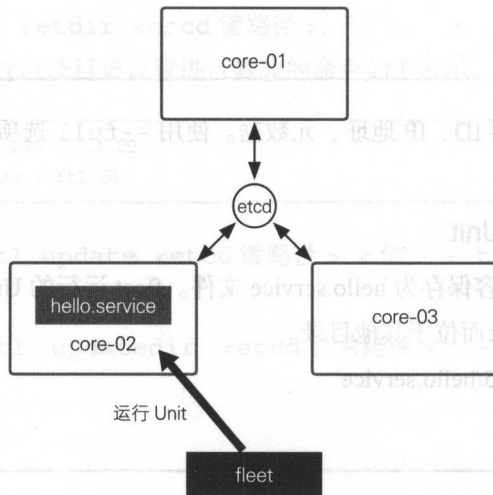


图 14-22 用 fleet 运行 Unit

使用 `fleetctl start` 命令运行 Unit 后, `hello.service` 文件中的内容会共享到集群的所有节点。

提示 其他 Unit 选项

在 Unit 文件中使用 `docker pull` 命令下载镜像时, 若花费很长时间, Unit 运行就会失败。如下所示, 将 `TimeoutStartSec` 设置为 0 后, 即使运行命令花费很长时间, Unit 创建也不会失败。

```
[Service]
TimeoutStartSec=0
```

Unit 非正常终止时, 下列选项用于自动重启。若将 `Restart` 设置为 `always`, 则将一直重启。向 `RestartSec` 指定重启前等待的时间, 如 `5min20s`、`100ms`、`10s` 等。

```
[Service]
Restart=always
RestartSec=5s
```

14.4.3 输出 fleet Unit 列表

运行如下命令, 输出集群中的 Unit 列表。

> core-01

```
$ fleetctl list-units
UNIT          DSTATE    TMACHINE          STATE    MACHINE          ACTIVE
hello.service launched d80aaff5.../172.17.8.102 launched d80aaff5.../172.17.8.102 active
```

由于前面运行了 `hello.serviceUnit`, 所以在 `core-02` (`d80aaff5, 172.17.8.102`) 中可以看到 `hello.serviceUnit` 运行的情形。根据不同情况, 各位运行 Unit 的节点可能不同。

14.4.4 查看 fleet Unit 状态

运行 `hello.service` Unit 的节点中, 运行如下命令。我的 `hello.service` Unit 是在 `core-02` 中运行的, 所以在 `core-02` 中运行如下命令。

> core-02

```
$ fleetctl status hello.service
• hello.service - Hello Service
  Loaded: loaded (/run/fleet/units/hello.service; linked-runtime)
  Active: active (running) since Mon 2014-09-08 13:30:42 UTC; 2min 43s ago
  Process: 1115 ExecStartPre=/usr/bin/docker rm hello (code=exited, status=1/FAILURE)
  Process: 1059 ExecStartPre=/usr/bin/docker kill hello (code=exited, status=1/FAILURE)
  Main PID: 1129 (docker)
  CGroup: /system.slice/hello.service
          └─1129 /usr/bin/docker run --name hello busybox /bin/sh -c while true; do echo Hello World;
sleep 1; done
```

```
Sep 08 13:33:16 core-02 docker[1129]: Hello World
Sep 08 13:33:17 core-02 docker[1129]: Hello World
Sep 08 13:33:18 core-02 docker[1129]: Hello World
Sep 08 13:33:19 core-02 docker[1129]: Hello World
Sep 08 13:33:20 core-02 docker[1129]: Hello World
Sep 08 13:33:21 core-02 docker[1129]: Hello World
Sep 08 13:33:22 core-02 docker[1129]: Hello World
Sep 08 13:33:23 core-02 docker[1129]: Hello World
Sep 08 13:33:24 core-02 docker[1129]: Hello World
Sep 08 13:33:25 core-02 docker[1129]: Hello World
```

命令格式为 `fleetctl status <Unit 名称>`。显示当前 Unit 的状态与日志。

使用 `fleetctl journal` 命令，可以只输出 Unit 的日志。

> core-02

```
$ fleetctl journal -f hello.service
```

命令格式为 `fleetctl journal <Unit 名称>`。使用 `-f` 选项实时输出日志。

14.4.5 测试 fleet 的自动恢复功能

某个节点发生故障时，fleet 会将其中运行的 Unit 迁移到其他节点运行。假设运行 `hello.service` Unit 的节点（我在 `core-02` 中运行）发生故障，则在其中运行如下命令终止虚拟机。

> core-02

```
$ sudo halt
```

接着，在其他节点中显示服务器（机器）列表。

> core-01

```
$ fleetctl list-machines
```

MACHINE	IP	METADATA
6960c4ce...	172.17.8.103	-
b3318c7b...	172.17.8.101	-

由于我终止了 `core-02`，故只显示 `core-01`（`b3318c7b`，`172.17.8.101`）与 `core-03`（`6960c4ce`，`172.17.8.103`）。

显示集群的 Unit 列表。

> core-01

```
$ fleetctl list-units
```

UNIT	DSTATE	TMACHINE	STATE	MACHINE	ACTIVE
------	--------	----------	-------	---------	--------

```
hello.service launched 6960c4ce.../172.17.8.103 launched 6960c4ce.../172.17.8.103 active
```

我的 `hello.serviceUnit` 在 `core-03` (6960c4ce, 172.17.8.103) 重新运行。根据不同情况, 各位运行 Unit 的节点可能不同。像这样, CoreOS 使用 `fleet` 提供高可用性 (High Availability)。

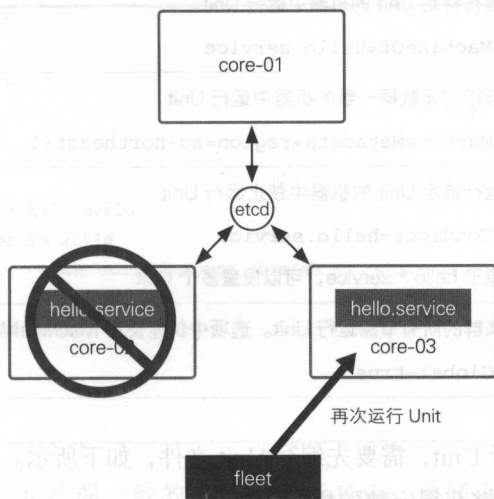


图 14-23 fleet Unit 自动恢复

提示 在 `vagrant` 中再次启动虚拟机

```
$ sudo vagrant halt -f core-02
$ sudo vagrant up core-02
```

确认 Unit 再次运行后, 停止 Unit (`hello.service`), 并从集群中删除。

> `core-01`

```
$ fleetctl stop hello.service
$ fleetctl destroy hello.service
```

14.4.6 使用 fleet 专用选项

目前为止, 集群中运行 Unit 的节点是任意选定的。下面学习有关 `fleet` 专用选项的内容, 利用其可以设置运行 Unit 的节点。

`systemd` Unit 文件 (`.service`) 中, `fleet` 专用选项使用 `[X-Fleet]` 部分。

表 15-1 fleet 专用选项

选项名称	说明
MachineID	在特定机器中运行 Unit。机器 ID 必须设置为完整形式，不能使用省略形式 如: MachineID=6960c4ce6f6b44518687a32d5e39dec9
MachineOf	在运行特定 Unit 的机器中运行 Unit 如: MachineOf=hello.service
MachineMetadata	在与指定元数据一致的机器中运行 Unit 如: MachineMetadata=region=ap-northeast-1
Conflicts	在运行特定 Unit 的机器中禁止运行 Unit 如: Conflicts=hello.service 类似于 hello.*.service，可以设置多个 Unit
Global	在集群的所有节点运行 Unit。选项中仅能使用 MachineMetadata，其他忽略 如: Global=true

要想在特定机器中运行 Unit，需要先编写 Unit 文件，如下所示。使用 `fleetctl list-machines -full` 命令能够获得完整的机器 ID。

➤ `dockerbook/Chapter15/hello.service`

> `~/hello.service`

```
[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello
ExecStartPre=/usr/bin/docker rm hello
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello

[X-Fleet]
MachineID=6960c4ce6f6b44518687a32d5e39dec9
```

使用 `fleetctl start` 命令运行 Unit 后，将在 `core-03` (6960c4ce, 172.17.8.103) 中运行。(每次安装 CoreOS 时机器 ID 都不同，所以本书中出现的机器 ID 与各位的机器 ID 可能不同。)

```
$ fleetctl start hello.service
Job hello.service launched on 6960c4ce.../172.17.8.103
```

下列示例将在运行着特定 Unit (hello.service) 的机器中运行 Unit。这种方式灵活应用于获取 Web 服务器、DB 等 IP 地址与端口的 sidekick 模型。

➤ dockerbook/Chapter15/world.service

> world.service

```
[Unit]
Description=World Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill world
ExecStartPre=/usr/bin/docker rm world
ExecStart=/usr/bin/docker run --name world busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop world

[X-Fleet]
MachineOf=hello.service
```

下列示例将在与指定元数据一致的机器中运行 Unit。地区为 ap-northeast-1，在云平台 amazon 机器中运行 Unit。

➤ dockerbook/Chapter15/hello-nginx.service

> hello-nginx.service

```
[Unit]
Description=Hello Nginx Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello-nginx
ExecStartPre=/usr/bin/docker rm hello-nginx
ExecStart=/usr/bin/docker run --name hello-nginx -p 80:80 nginx:latest
ExecStop=/usr/bin/docker stop hello-nginx

[X-Fleet]
MachineMetadata="region=ap-northeast-1" "provider=amazon"
```

若想在 us-east-1 与 us-west-1 中运行 Unit，则要进行如下设置。

```
[X-Fleet]
MachineMetadata=region=us-east-1
MachineMetadata=region=us-west-1
```

下列示例将在运行特定 Unit 的机器中禁止运行 Unit，而在其他机器中运行。存在 Web 服务器 Unit 与 DB Unit 时，DB Unit 将在不允许 Web 服务器 Unit 的机器中运行。

> dockerbook/Chapter15/db.service

```
> db.service
[Unit]
Description=DB Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill db
ExecStartPre=/usr/bin/docker rm db
ExecStart=/usr/bin/docker run --name db -p 27017:27017 mongo:latest
ExecStop=/usr/bin/docker stop db

[X-Fleet]
Conflicts=web.*.service
```

此处的 web.*.service 表示 web.1.service、web.2.service……。

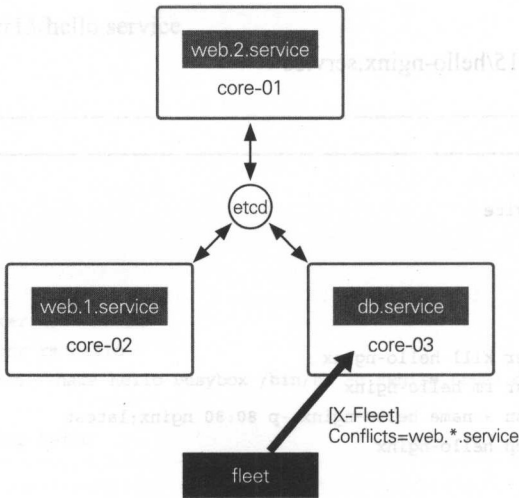


图 14-24 使用 X-Fleet Conflicts 选项运行 Unit

14.4.7 灵活使用 fleet Unit 文件模板

systemd 提供了模板功能，借助该功能可以使用 1 个 Unit 文件创建多个实例。在此过程中灵活使用 fleet 可以使用 1 个 Unit 文件在多个节点中创建 Unit。

► Unit 文件模板的文件名格式为 <Unit 名称>@.<扩展名>。如 hello@.service。

► 使用 Unit 文件模板创建 Unit 时采用的格式为 <Unit 名称>@<实例名>.<扩展名>。如 hello@world.service、hello@1.service。

```

      %p
    ┌───┴───┐
hello@1.service
    └───┬───┘
      %i
  
```

图 14-25 fleet Unit 文件模板

在 core-01 中将如下内容保存为 hello@.service 文件。

► dockerbook/Chapter15/hello@.service

> ~/hello@.service

```

[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello-%i
ExecStartPre=/usr/bin/docker rm hello-%i
ExecStart=/usr/bin/docker run --name hello-%i busybox /bin/sh -c "while true; do echo Hello World;
sleep 1; done"
ExecStop=/usr/bin/docker stop hello-%i

[X-Fleet]
Conflicts=hello@*.service
  
```

► %i：实例名。代入 @ 后面出现的实例名。

► Conflicts：设置为 hello@*.service，分别在不同节点运行 Unit，防止重复。

运行如下命令，将 hello@.service 文件保存到集群，然后输出集群中存储的 Unit 文件列表。

> core-01

```

$ fleetctl submit hello@.service
$ fleetctl list-unit-files
UNIT          HASH      DSTATE
hello@.service 1aeee55 inactive
  
```

► 命令格式为 fleetctl submit <Unit 文件>。不运行 Unit，只将 Unit 文件的内容保

存到集群。

➤ `fleetctl list-unit-files` 命令用于输出集群中存储的 Unit 文件列表。示例显示存储在集群中的 `hello@.service` 文件。

接下来，执行如下命令，运行 Unit。

> **core-01**

```
$ fleetctl start hello@{1..3}.service
Unit hello@1.service launched on d80aaff5.../172.17.8.102
Unit hello@2.service launched on 6960c4ce.../172.17.8.103
Unit hello@3.service launched on b3318c7b.../172.17.8.101
```

命令格式为 `fleetctl start <Unit 名称>@{ 起始数字...结束数字 }.service`。如上所示，输入 `{1..3}` 则创建 1、2、3 这 3 个实例。使用 `fleetctl start hello@1.service` 命令也可以单独创建 Unit。执行命令后，Unit 不会集中到特定节点，而是分别在 `core-01`、`core-02`、`core-03` 中进行创建。

提示 systemd 标识符

以 `hello@1.service` 运行 `hello@.service` Unit 时：

- `%n`: Unit 文件完整名称。hello@1.service
- `%N`: 除扩展名之外的 Unit 文件名。hello@1
- `%p`: 出现在 @ 之前的 Unit 名称。hello
- `%i`: 出现在 @ 之后的实例名称。1

14.4.8 灵活使用 fleet sidekick 模型

在 CoreOS 集群中使用 fleet，可以根据机器 ID、元数据、是否运行特定 Unit 选择运行 Unit 的节点。也就是说，由于不是通过各节点的 IP 地址运行 Unit，故无法知道 Unit 的 IP 地址。此时，获取 IP 地址的方式就是 sidekick 模型。

提示 sidekick 模型

sidekick 一词是“助手”的意思。由于要一起运行主容器与助手容器以获取 IP 地址，故称为 sidekick 模型。

如前所述，使用 `etcd` 可以使各节点共享数据。而 sidekick 模型则使用 `etcd` 共享 IP 地址。在 `core-01` 中将如下内容保存为 `web.service` 文件。

➤ dockerbook/Chapter15/web.service

> ~/web.service

```
[Unit]
Description=Web Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill web
ExecStartPre=/usr/bin/docker rm web
ExecStart=/usr/bin/docker run --name web -p 80:80 nginx:latest
ExecStop=/usr/bin/docker stop web
```

将如下内容保存为 web-discovery.service 文件。

➤ dockerbook/Chapter15/web-discovery.service

> ~/web-discovery.service

```
[Unit]
Description=Announce Web
BindsTo=web.service

[Service]
EnvironmentFile=/etc/environment
ExecStart=/bin/sh -c \
"while true; \
do \
    etcdctl set /services/web/nginx \
    '{ \"host\": \"${COREOS_PUBLIC_IPV4}\", \"port\": 80 }' \
    --ttl 60; \
    sleep 45; \
done"
ExecStop=/usr/bin/etcdctl rm /services/web/nginx

[X-Fleet]
MachineOf=web.service
```

- **BindsTo**: 指定 web.service, web.serviceUnit 终止时, 也终止当前 Unit。
- **EnvironmentFile**: 该文件用于设置当前 Unit 的环境变量。由于 /etc/environment 文件存储着当前节点的公共 IP 地址 (COREOS_PUBLIC_IPV4) 与私有 IP 地址 (COREOS_PRIVATE_IPV4), 所以一定要设置。
- **ExecStart**: 运行 shell 脚本, 用于向 etcd 注册 IP 地址与端口号。
- 使用 etcdctl set 命令在 /services/web/nginx 中创建键, 然后保存 IP 地址与端口号。IP 地址存储于 COREOS_PUBLIC_IPV4 变量。

» 使用 `--ttl` 选项, 设置为 “60 秒后删除键”。

» `sleep 45` 用于每隔 45 秒执行 1 次 `etcdctl set` 命令, 并在删除键之前更新内容。正常节点可以继续持有键, 而中断节点则执行 `etcdctl set` 命令以删除键, 所以可以确切得知节点的状态。此外, 在这样反复执行的过程中, 当前节点中断后在其他节点中再次运行 Unit 时, 新的 IP 地址也可以更新到 `etcd`。

» `ExecStop`: 若 Unit 停止, 则删除 `etcd` 的 `/services/web/nginx` 键。

» `MachineOf`: 指定 `web.service`, 在运行 `web.serviceUnit` 的节点中运行当前 Unit。

使用 `fleetctl start` 命令, 运行 `web.service`、`web-discovery.serviceUnit`。

> core-01

```
$ fleetctl start web.service
Job web.service launched on d80aaff5.../172.17.8.102
$ fleetctl start web-discovery.service
Job web-discovery.service launched on d80aaff5.../172.17.8.102
```

在我的环境下, `web.service`、`web-discovery.serviceUnit` 运行于 `core-02` (`d80aaff5`, `172.17.8.102`)。

接下来, 使用 `etcdctl get` 命令输出 IP 地址与端口号。

> core-01

```
$ etcdctl get /services/web/nginx
{ "host": "172.17.8.102", "port": 80 }
```

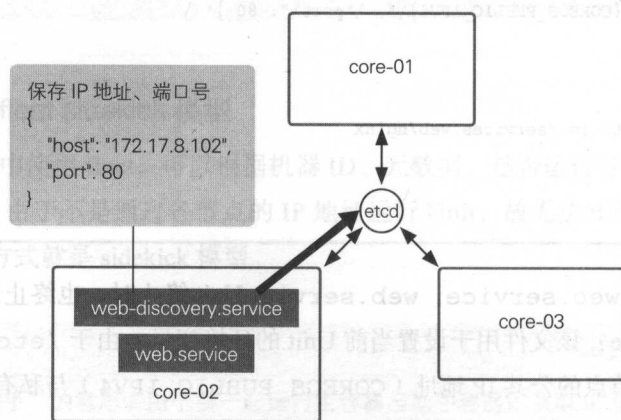


图 14-26 用 fleet sidekick 模型获取 IP 地址

14.4.9 fleet 其他命令

从集群中删除 Unit 文件的命令如下所示。

```
$ fleetctl destroy hello.service
```

命令格式为 `fleetctl destroy <Unit 名称>`。从集群中删除 Unit 文件后，所有节点中的 Unit 文件都会删除。

使用如下命令可以输出集群中存储的 Unit 文件的内容。

```
$ fleetctl cat hello.service
```

命令格式为 `fleetctl cat <Unit 名称>`。输出的是存储在集群中的 Unit 文件内容，而不是存储在本地的 Unit 文件。

下列命令只用于将 Unit 分配到节点而不运行。

```
$ fleetctl load hello.service
```

```
$ fleetctl list-units
```

UNIT	DSTATE	TMACHINE	STATE	MACHINE	ACTIVE
hello.service	loaded	6960c4ce.../172.17.8.103	loaded	6960c4ce.../172.17.8.103	inactive

命令格式为 `fleetctl load <Unit 名称>`。使用 `fleetctl list-units` 命令输出 Unit 列表，可以看到 `hello.service` 只是分配到节点而并未运行，所以其状态为 `inactive`。

从节点删除 Unit 的命令如下所示。

```
$ fleetctl unload hello.service
```

```
$ fleetctl list-units
```

UNIT	DSTATE	TMACHINE	STATE	MACHINE	ACTIVE
hello.service	inactive	-	inactive	-	-

命令格式为 `fleetctl unload <Unit 名称>`。使用 `fleetctl list-units` 命令输出 Unit 列表，可以看到 `hello.service` 并未分配到任何节点。与 `fleetctl destroy` 命令不同，该命令并不会将 Unit 文件从集群中删除。

14.5 在云服务中使用 CoreOS

14.5.1 在 Amazon EC2 中使用 CoreOS

下面学习在 Amazon Web Services 的 EC2 中使用 CoreOS 的方法。在 AWS 控制台创建 EC2 实

例，单击 1.Choose AMI 中左侧 AWS Marketplace 选项卡，向搜索窗口输入 coreos，然后选择 CoreOS。

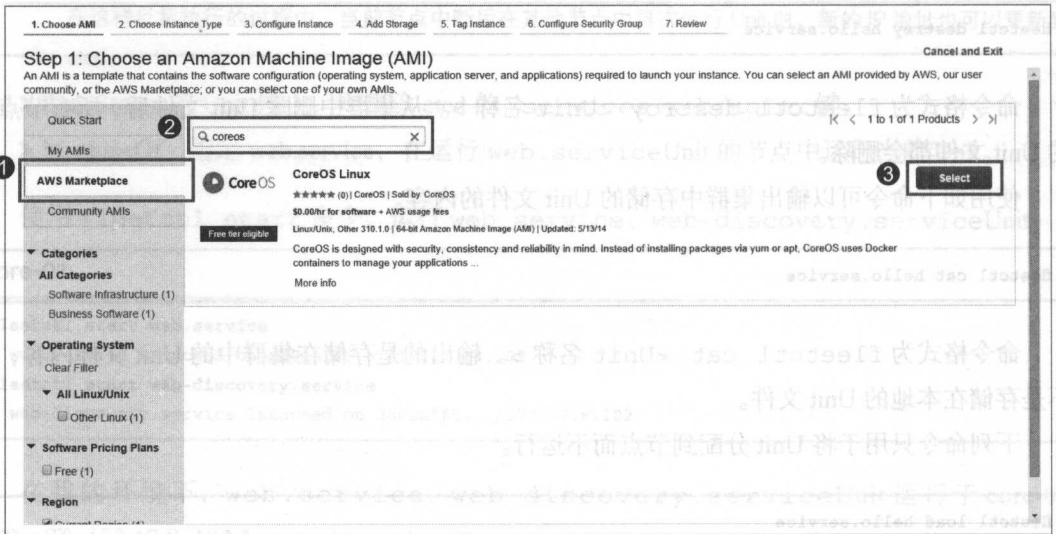


图 14-27 在 AWS Marketplace 中搜索 CoreOS

在 3.Configure Instance 中单击 Advanced Details，设置 User data，如下所示。

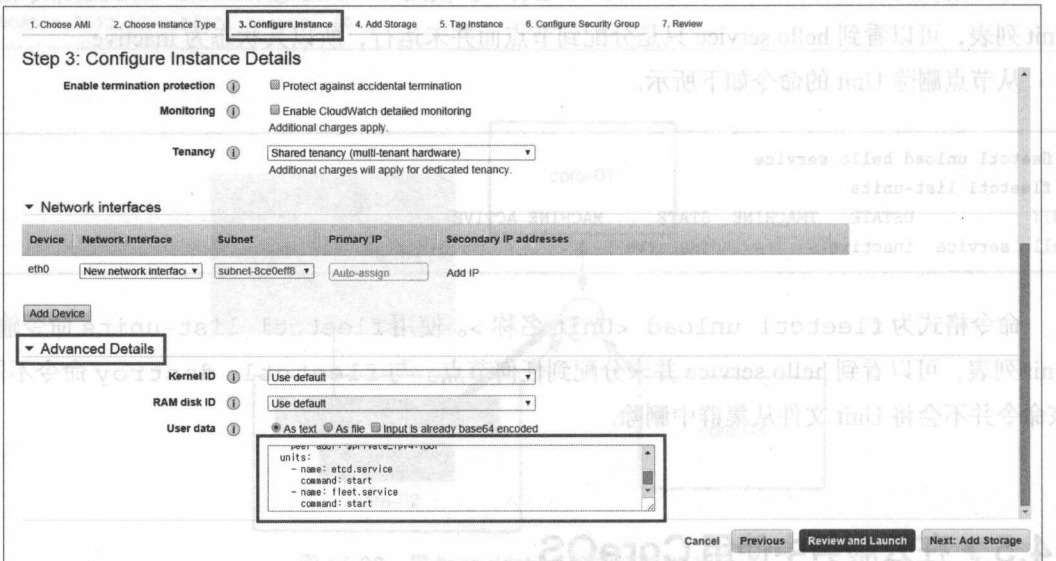


图 14-28 设置 CoreOS 实例 User data

在 User data 部分输入如下内容。

➤ dockerbook/Chapter15/ec2/user-data

```
#cloud-config
coreos:
  etcd:
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    discovery: https://discovery.etcd.io/<集群 ID>
    # multi-region and multi-cloud deployments need to use $public_ipv4
    addr: $private_ipv4:4001
    peer-addr: $private_ipv4:7001
  units:
    - name: etcd.service
      command: start
    - name: fleet.service
      command: start
```

- discovery: 设置 discovery URL。申请分配 discovery URL 的方法请参考 15.2 节。
- addr、peer-addr: 设置为 \$private_ipv4, 将自动设置 IP 地址。若想在多个地区创建实例或者与其他云服务的 CoreOS 一起使用, 不要设置为 \$private_ipv4, 只要设置为 \$public_ipv4 即可。

14.5.2 在 Google Compute Engine 中使用 CoreOS

Google Cloud Platform 的 Compute Engine 中也可以使用 CoreOS。创建 VM 实例时, 要想使用 cloud-config.yaml 文件, 需要安装 Google Cloud SDK。关于安装 Google Cloud SDK 的方法, 请参考 11.1 节。

将如下内容保存为 cloud-config.yaml 文件。

➤ dockerbook/Chapter15/gce/cloud-config.yaml

> cloud-config.yaml

```
#cloud-config
coreos:
  etcd:
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    discovery: https://discovery.etcd.io/<集群 ID>
    # multi-region and multi-cloud deployments need to use $public_ipv4
    addr: $private_ipv4:4001
    peer-addr: $private_ipv4:7001
  units:
    - name: etcd.service
      command: start
```

```

- name: fleet.service
  command: start

```

- discovery: 设置 discovery URL。申请分配 discovery URL 的方法请参考 15.2 节。
- addr、peer-addr: 设置为 \$private_ipv4，将自动设置 IP 地址。若想在多个地区创建实例或者与其他云服务的 CoreOS 一起使用，不要设置为 \$private_ipv4，只要设置为 \$public_ipv4 即可。

输入如下命令，创建 3 个 CoreOS VM 实例。在 Enter passphrase、Enter same passphrase again 中全部按 Enter 键。

```

$ gcutil --project=<谷歌云项目 ID> addinstance \
--image=projects/coreos-cloud/global/images/coreos-alpha-435-0-0-v20140910 \
--persistent_boot_disk \
--zone=asia-east1-a \
--machine_type=f1-micro \
--metadata_from_file=user-data:cloud-config.yaml \
core-01 core-02 core-03

```

WARNING: You don't have an ssh key for Google Compute Engine. Creating one now...

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

- --project: 输入谷歌云项目 ID。
- --image: 镜像名称。设置为 projects/coreos-cloud/global/images/coreos-alpha-435-0-0-v20140910。若镜像不存在，则到下列 URL 的 Choosing a Channel 中查看最新版本。
<https://coreos.com/docs/running-coreos/cloud-providers/google-compute-engine/>
- --persistent_boot_disk: 使用启动盘。
- --zone: 要创建 VM 实例的地区。我设置为 asia-east1-a。
- --machine-type: VM 实例机类型。我设置为 f1-micro。
- --metadata-from_file: 设置元数据文件。设置为 user-data:cloud-config.yaml。
- VM 实例名称分别设置为 core-01、core-02、core-03。

图 14-28 部署 CoreOS 实例

在 User data 部分输入如下内容

第 15 章

DOCKER

使用 Docker 搭建 WordPress 博客

WordPress 是开源博客系统。虽然 Docker Hub 中有官方镜像，但本章将学习如何使用 Docker 直接搭建。

首先创建 WordPress 镜像与数据库镜像。

- ▶ WordPress 镜像：安装要用作 Web 服务器的 Apache。由于 WordPress 使用 PHP 语言编写，所以也要安装 PHP。
- ▶ 数据库镜像：由于 WordPress 要使用 MySQL 数据库，所以也要安装 MySQL。

为了能够在 WordPress 容器中使用数据库容器，创建容器时，在 `docker run` 命令中使用 `--link` 选项进行连接。



图 15-1 使用 Docker 搭建 WordPress 博客

请从我的 GitHub 仓库下载示例文件。

- ▶ <https://github.com/pyrasis/dockerbook>

15.1 > 编写 WordPress Dockerfile 文件

首先创建 WordPress Docker 镜像。创建 wordpress 目录后，将如下内容保存为 Dockerfile 文件。

> dockerbook/Chapter16/wordpress/Dockerfile

```
~$ mkdir wordpress
```

```
~$ cd wordpress
```

> ~/wordpress/Dockerfile

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y apache2 php5 php5-mysql mysql-client wget
```

```
WORKDIR /var/www
```

```
RUN wget http://ko.wordpress.org/wordpress-4.0-ko_KR.tar.gz -O - | tar -xz
```

```
WORKDIR /etc/apache2/sites-enabled
```

```
RUN sed -i "s/\/var\/www\/html\/\/var\/www\/
```

```
RUN mv wp-config-sample.php wp-config.php
```

```
RUN sed -i "s/'database_name_here'/'wp'/g" wp-config.php && \
```

```
sed -i "s/'username_here'/'root'/g" wp-config.php && \
```

```
sed -i "s/'password_here'/'getenv('DB_ENV_MYSQL_ROOT_PASSWORD')'/g" wp-config.php && \
```

```
sed -i "s/'localhost'/'db'/g" wp-config.php
```

```
ADD entrypoint.sh /entrypoint.sh
```

```
RUN chmod +x /entrypoint.sh
```

```
ENTRYPOINT /entrypoint.sh
```

我在 Ubuntu 14.04 中使用 apt-get 工具安装所需的包。

- > 设置使用 FROM 基于 ubuntu: 14.04 创建镜像。
- > 使用 apt-get update 将包列表更新为最新状态，然后安装 apache2、php5、php5-mysql、mysql-client、wget。
- > 使用 wget 将 WordPress 源文件下载到 /var/www 目录，然后解压缩。
- > 使用 sed 修改 /etc/apache2/sites-enabled 目录 000-default.conf 文件的内容。将 Web 服务器的默认目录从 /var/www/html 修改为 /var/www/wordpress，以使用 WordPress 源文件。
- > 将 /var/www/wordpress 目录 wp-config-sample.php 文件名修改为 wp-config.php，然后使用 sed 修改 DB 设置。

- » 设置 DB_NAME 为 wp。
- » 设置 DB_USER 为 root。
- » 使用环境变量的 DB_ENV_MYSQL_ROOT_PASSWORD 设置 DB_PASSWORD。使用 docker run 命令的 --link 选项连接容器时，所连接容器的环境变量格式为 < 别名 >_ENV_< 环境变量 >。由于连接容器时将别名设置为 db，数据库容器要使用的环境变量为 MYSQL_ROOT_PASSWORD，所以最后名称为 DB_ENV_MYSQL_ROOT_PASSWORD。
- » 由于连接容器时指定的别名为 db，所以将 DB_HOST 设置为 db。
- » 设置权限，使添加 entrypoint.sh 文件后可以运行。
- » 将 ENTRYPOINT 设置为 /entrypoint.sh 文件，容器启动时运行脚本文件。

将如下内容保存为 entrypoint.sh 文件。

- » dockerbook/Chapter16/wordpress/entrypoint.sh

> ~/wordpress/entrypoint.sh

```
#!/bin/sh

mysql -h db -uroot -p$DB_ENV_MYSQL_ROOT_PASSWORD -e "create database wp"
apachectl -DFOREGROUND
```

- » WordPress 必须事先创建 MySQL 数据库。使用 mysql 命令连接 db 后，创建 wp 数据库。使用的用户账号为 root，密码为环境变量的 DB_ENV_MYSQL_ROOT_PASSWORD。
- » 以 foreground 方式运行 Apache 网页服务器。请注意，若不以 foreground 方式运行 Apache，则即使使用 docker run -d 创建容器也会直接停止。

使用 docker build 命令创建镜像。

```
~/wordpress$ sudo docker build --tag wordpress .
```

15.2 » 编写 MySQL 数据库 Dockerfile 文件

下面创建数据库镜像。先创建 mysql 目录，然后将如下内容保存为 Dockerfile 文件。

- » dockerbook/Chapter16/mysql/Dockerfile

```
~$ mkdir mysql
~$ cd mysql
```

> ~/mysql/Dockerfile

```
FROM ubuntu:14.04

ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update
RUN echo "mysql-server mysql-server/root_password password" | debconf-set-selections
RUN echo "mysql-server mysql-server/root_password_again password" | debconf-set-selections
RUN apt-get install -y mysql-server

WORKDIR /etc/mysql
RUN sed -i "s/127.0.0.1/0.0.0.0/g" my.cnf

ADD entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
EXPOSE 3306

ENTRYPOINT /entrypoint.sh
```

- 必须使用 ENV 命令将 DEBIAN_FRONTEND 环境变量设置为 noninteractive。若使用 apt-get 安装 MySQL 包，则会出现用户之间输入 root 密码的部分。但是，由于创建 Docker 镜像时无法输入，所以必须设置为 noninteractive 直接跳过，用户不必输入。
- 使用 apt-get update 将包目录更新为最新状态。
- 将 mysql-server mysql-server/root_password password 设置到 debconf-set-selections，在用 noninteractive 跳过的部分应用密码设置。也可以在 password 后面输入实际要使用的密码，但由于要在 docker run 命令中使用 -e 选项设置密码，故不进行任何输入。
- mysql-server mysql-server/root_password_again password 同上。
- 使用 apt-get install 安装 mysql-server 包。
- 使用 sed 修改 /etc/mysql 目录 my.cnf 文件的内容。将 bind-address=127.0.0.1 修改为 bind-address=0.0.0.0。若不修改，则无法从外部连接 MySQL。
- 设置权限，使添加 entrypoint.sh 文件之后可以运行。
- 将 EXPOSE 设置为 3306，使可以连接 3306 端口。
- 将 ENTRYPOINT 设置为 /entrypoint.sh 文件，容器启动时运行脚本文件。

提示 VOLUME

根据不同情况，也可以不将数据库保存到容器，而将 VOLUME 设置为存储到主机。

```
VOLUME ["/var/lib/mysql"]
```

将如下内容保存为 `entrypoint.sh` 文件。

➤ `dockerbook/Chapter16/mysql/entrypoint.sh`

> `~/mysql/entrypoint.sh`

```
#!/bin/bash

if [ -z $MYSQL_ROOT_PASSWORD ]; then
    exit 1
fi

mysql_install_db --user mysql > /dev/null

cat > /tmp/sql <<EOF
USE mysql;
FLUSH PRIVILEGES;
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION;
UPDATE user SET password=PASSWORD("$MYSQL_ROOT_PASSWORD") WHERE user='root';
EOF

mysqld --bootstrap --verbose=0 < /tmp/sql
rm -rf /tmp/sql

mysqld
```

- 若环境变量中无 `MYSQL_ROOT_PASSWORD`，则不运行数据库并退出。
- 使用 `mysql_install_db` 安装数据库文件。
- 将设置 MySQL root 账户密码的 SQL 语句保存为 `/tmp/sql` 文件。密码使用环境变量的 `MYSQL_ROOT_PASSWORD` 中存储的值。之所以不在 Dockerfile 中设置密码而在此处设置密码，是为了使用 `docker run` 命令的 `-e` 选项设置密码。使用 `-e` 选项设置的环境变量值只能在 CMD、ENTRYPOINT 中使用。
- 向 `mysqld` 设置 `--bootstrap` 选项，输入 `/tmp/sql` 文件，设置 root 账户密码。然后删除 `/tmp/sql` 文件。
- 最后，运行 `mysqld`。与 Apache 一样，也要以 foreground 形式运行 MySQL。

使用 `docker build` 命令创建镜像。

```
~/mysql$ sudo docker build --tag mysql .
```

> <https://github.com/empyris/dockerbook>

15.3 > 创建 WordPress 与数据库容器

准备 WordPress 与数据库镜像后，创建容器。

```
$ sudo docker run -d --name db -e MYSQL_ROOT_PASSWORD=examplepassword mysql
$ sudo docker run -d --name example-wp -p 80:80 --link db:db wordpress
```

- ▶ 创建数据库容器时使用 `-e` 选项，向 `MYSQL_ROOT_PASSWORD` 设置要使用的 root 账号的密码。
- ▶ 创建 WordPress 容器时，使用 `--link` 选项连接 db 容器与 db 别名。然后使用 `-p` 选项使得从外部可以访问 80 端口。

容器创建完毕后，运行 Web 浏览器，访问服务器的 IP 地址或域名。

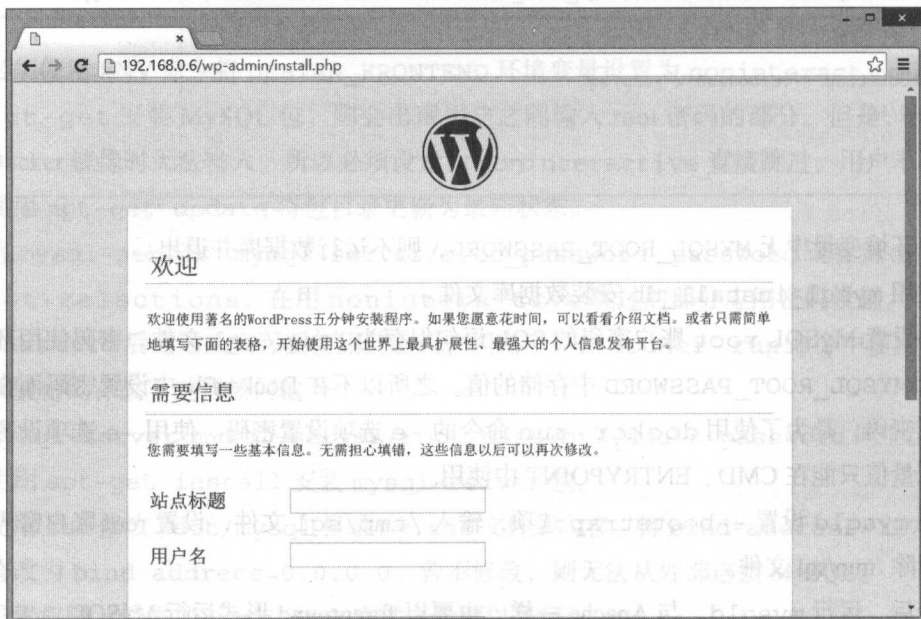


图 15-2 在 Web 浏览器中访问 WordPress 容器

出现 WordPress 安装界面。进行基本设置即可使用 WordPress 博客系统。

第 16 章

DOCKER

使用 Docker 构建 Ruby on Rails 应用

Ruby on Rails 是使用 Ruby 编写的开源 Web 框架。本章将学习使用 Docker 构建 Ruby on Rails 应用的方法。

创建 Docker 镜像前，首先搭建 Ruby on Rails 开发环境。

- 安装 Git 及所需的包。
- 安装 rbenv。
- 使用 rbenv 安装 Ruby。
- 使用 gem 安装 Rails、Unicorn。

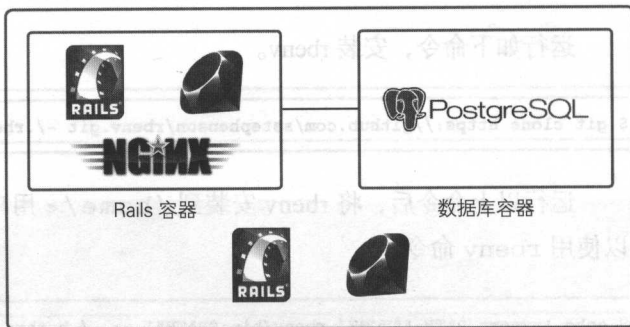


图 16-1 使用 Docker 构建 Rails 应用

创建 Rails 镜像与数据库镜像。

- Rails 镜像：安装要用作 Web 服务器的 Nginx。使用 rbenv 安装 Ruby 及所需的 gem (Rails、Unicorn 等)。
- 数据库镜像：安装 PostgreSQL。MySQL 的安装方法请参考 16.2 节。

为了能够在 Rails 容器中使用数据库容器，创建容器时要在 `docker run` 命令中使用 `--link` 选项进行连接。

请从我的 GitHub 仓库下载示例文件。

- <https://github.com/pyrasis/dockerbook>

16.1 > 安装 Ruby 与 Rails

创建 Rails 镜像前，必须先搭建 Ruby on Rails 开发环境。下面将不使用各 Linux 发布版的包，而使用 rbenv 安装 Ruby。

首先安装必需的包。

> Ubuntu

```
$ sudo apt-get install autoconf bison build-essential libssl-dev libyaml-dev libreadline6-dev zlib1g-dev libncurses5-dev git
```

> CentOS

```
$ sudo yum install gcc openssl-devel libyaml-devel readline-devel zlib-devel git
```

运行如下命令，安装 rbenv。

```
$ git clone https://github.com/sstephenson/rbenv.git ~/.rbenv
```

运行以上命令后，将 rbenv 安装到 /home/<用户账号>/.rbenv 目录。运行如下命令，以使用 rbenv 命令。

```
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
$ source ~/.bashrc
```

安装 ruby-build 以使用 rbenv install 命令。安装 rbenv-gem-rehash 后，则安装 gem 时不必每次都输入 rbenv rehash 命令。

```
~$ mkdir ~/.rbenv/plugins
~$ cd ~/.rbenv/plugins
~/.rbenv/plugins$ git clone https://github.com/sstephenson/ruby-build.git
~/.rbenv/plugins$ git clone https://github.com/sstephenson/rbenv-gem-rehash.git
```

接着转到用户目录，然后使用 rbenv install 命令安装 Ruby 2.1.3 版本。使用 rbenv local 命令设置为只有当前用户才能使用 Ruby。

```
~/.rbenv/plugins$ cd
~$ rbenv install 2.1.3
~$ rbenv local 2.1.3
```

提示 查看可以安装的 Ruby 版本

运行如下命令，显示可以安装的 Ruby 版本。请各位根据情况进行选择。

```
$ rbenv install -l
```

使用 gem 命令，安装 Rails、Unicorn。

```
$ gem install rails unicorn
```

由于计划使用 PostgreSQL 数据库，故需要安装 PostgreSQL gem 包。

> Ubuntu

```
$ sudo apt-get install libpq-dev
```

> CentOS

```
$ sudo yum install postgresql-devel
```

提示 使用 MySQL**> Ubuntu**

```
$ sudo apt-get install libmysqlclient-dev
```

> CentOS

```
$ sudo yum install mysql-devel
```

由于部分 gem 使用 JavaScript 编写，所以也要安装 Node.js。

> Ubuntu

```
$ sudo apt-get install nodejs
```

> CentOS 6

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ sudo yum install nodejs
```

> CentOS 7

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
$ sudo yum install nodejs
```

提示 CentOS7 EPEL 包版本

CentOS 7 EPEL 包版本升级速度快。若无法下载 rpm 文件，请先访问 http://dl.fedoraproject.org/pub/epel/7/x86_64/e/ 检查有无新版本，再使用 yum 命令安装相应版本。

16.2 ▸ 编写 Rails Dockerfile

安装 Ruby 与 Rails 后，开始创建 Rails 应用。

```
~$ rails new exampleapp --database=postgresql
```

创建 exampleapp 目录。打开 exampleapp/config 目录 database.yml 文件，进行如下修改。

▸ dockerbook/Chapter17/exampleapp/config/database.yml

▸ ~/exampleapp/config/database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see rails configuration guide
  # http://guides.rubyonrails.org/configuring.html#database-pooling
  pool: 5
  template: template0
  username: postgres
  password: <%= ENV['DB_ENV_POSTGRES_PASSWORD'] %>
  host: <%= ENV['POSTGRES_HOST'] || 'db' %>
```

- **template**: 设置为 template0。在 PostgreSQL 中创建数据库时会复制已有数据库。template0 可以设置新的编码方式，若不进行设置，则会引发 UTF-8 错误。
- **username**: 设置为 postgres。
- **password**: 设置使用环境变量的 DB_ENV_POSTGRES_PASSWORD。使用 docker run 命令的 --link 选项连接容器时，所连接容器的环境变量格式为 <别名>_ENV_<环境变量>。由于连接容器时将别名设置为 db，并且要在数据库容器中使用 POSTGRES_PASSWORD 环境变量，所以最后得到的完整形式为 DB_ENV_POSTGRES_PASSWORD。
- **host**: 使用 || 运算符，分别设置要在开发环境与数据库容器中使用的数据库主机。开发时，向环境变量的 POSTGRES_HOST 设置数据库容器的 IP 地址。另外，由于连接数据库容器时要将别名指定为 db，故此处设置为 db。

提示 使用 MySQL

```
~$ rails new exampleapp --database=mysql
```

- dockerbook/Chapter17/exampleapp_mysql/config/database.yml

```
> ~/exampleapp/config/database.yml
```

```
default: &default
  adapter: mysql2
  encoding: utf8
  pool: 5
  username: root
  password: <%= ENV['DB_ENV_MYSQL_ROOT_PASSWORD'] %>
  host: <%= ENV['MYSQL_HOST'] || 'db' %>
```

为了使用 Unicorn, 打开 exampleapp 目录的 Gemfile, 删除 gem 'unicorn' 部分的注释。

```
> dockerbook/Chapter17/exampleapp/Gemfile
```

```
> ~/exampleapp/Gemfile
```

```
gem 'unicorn'
```

将如下内容保存为 Dockerfile 文件。

```
> dockerbook/Chapter17/exampleapp/Dockerfile
```

```
> ~/exampleapp/Dockerfile
```

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y autoconf bison build-essential \
  libssl-dev libyaml-dev libreadline6-dev zlib1g-dev libncurses5-dev git
```

```
RUN apt-get install -y nginx nodejs curl libpq-dev
```

```
RUN git clone https://github.com/sstephenson/rbenv.git /root/.rbenv
```

```
RUN git clone https://github.com/sstephenson/ruby-build.git \
  /root/.rbenv/plugins/ruby-build
```

```
ENV PATH /root/.rbenv/bin:/root/.rbenv/shims:$PATH
```

```
ENV CONFIGURE_OPTS --disable-install-doc
```

```
RUN rbenv install 2.1.3
```

```
RUN rbenv global 2.1.3
```

```
RUN rbenv init -
```

```
RUN echo 'gem: --no-rdoc --no-ri' >> /root/.gemrc
```

```
RUN gem install bundler
```

```
RUN rbenv rehash
```

```

RUN echo "daemon off;" >> /etc/nginx/nginx.conf
RUN rm -rf /etc/nginx/sites-enabled/default
ADD exampleapp.conf /etc/nginx/sites-enabled/exampleapp.conf

WORKDIR /tmp
ADD Gemfile Gemfile
ADD Gemfile.lock Gemfile.lock
RUN bundle install

ADD ./ /var/www/exampleapp
WORKDIR /var/www/exampleapp
RUN chmod +x entrypoint.sh

EXPOSE 80

ENTRYPOINT ./entrypoint.sh

```

在 Ubuntu 14.04 中使用 apt-get 工具安装所需的包。

- ▶ 使用 FROM 指令设置基于 ubuntu: 14.04 创建镜像。
- ▶ 使用 apt-get update 将包列表更新至最新状态，然后安装 Git 与 Ruby 所需的包。此外，也要安装 nginx、nodejs、libpq-dev。
 - » 需要安装 nodejs，以运行由 JavaScript 编写的 gem。
 - » 需要安装 libpq-dev，以安装 PostgreSQL gem。
- ▶ 使用 git 命令，将 rbenv 下载到 /root/.rbenv 目录。
- ▶ 使用 git 命令，将 ruby-build 下载到 /root/.rbenv/plugins/ruby-build 目录中。
- ▶ 使用 ENV 命令，将 rbenv 路径添加到环境变量 PATH。
- ▶ 使用 rbenv 命令安装 Ruby。
 - » 使用 ENV 命令向 CONFIGURE_OPTS 环境变量添加 --disable-install-doc 选项，不安装文档。
 - » 使用 rbenv install 命令安装 Ruby 2.1.3 版本。
 - » 使用 rbenv global 命令设置 Ruby 2.1.3 版本在所有账户可用。
 - » 运行 rbenv init - 命令，初始化 rbenv。
- ▶ 安装 bundler。
 - » 向 .gemrc 文件添加 --no-rdoc、--no-ri 选项，安装 gem 时，不安装文档与 ri（文档工具）。
 - » 使用 gem 命令安装 bundler，运行 rbenv rehash 命令。
- ▶ 设置以 foreground 方式（非守护进程方式）运行 Nginx。若以守护进程方式运行 Nginx，Docker 容器会立即终止，请各位注意。
- ▶ 删除 /etc/nginx/sites-enabled 目录的 nginx 默认设置文件（default），添加 exampleapp.conf 文件。

- 向 /tmp 目录添加 Gemfile、Gemfile.lock 文件，然后使用 `bundle install` 命令安装 gem 文件。
- 将 Rails 应用程序目录添加到 /var/www/exampleapp 目录。
- 设置权限，使可以运行 `entrypoint.sh` 文件。
- 设置 EXPOSE 为 80，以连接 80 号端口。
- 将 ENTRYPOINT 设置为 `/entrypoint.sh` 文件，容器启动时运行脚本文件。

提示 缩短 Docker 镜像创建时间

如下所示，添加 Rails 应用程序目录（exampleapp）后，若运行 `bundle install` 命令，则每当 Rails 应用程序的 .rb 文件或其他文件发生更改时，都要重新安装 gem 文件。因为 Dockerfile 中，若使用 ADD 命令添加的目录下的文件发生变化，则会再次执行其后的命令。

```
ADD ./ /var/www/exampleapp
WORKDIR /var/www/exampleapp
RUN bash -l -c "bundle install"
```

如下所示，添加 Rails 应用程序目录前，只单独添加 Gemfile、Gemfile.lock 文件，然后再运行 `bundle install` 命令。因此，即使 .rb 文件等发生变化，也不会重新安装 gem 文件。

```
WORKDIR /tmp
ADD Gemfile Gemfile
ADD Gemfile.lock Gemfile.lock
RUN bundle install
```

```
ADD ./ /var/www/exampleapp
```

为了灵活使用 Dockerfile 的缓存功能，通常将不常更改且耗时较长的部分单独分离并上移。

提示 使用 MySQL

- dockerbook/Chapter17/exampleapp_mysql/Dockerfile

➤ ~/exampleapp/Dockerfile

```
RUN apt-get install -y nginx nodejs curl libmysqlclient-dev
```

需要 Unicorn 设置文件。将如下内容保存为 unicorn.rb 文件。

➤ dockerbook/Chapter17/exampleapp/unicorn.rb

➤ ~/exampleapp/unicorn.rb

```
worker_processes 4
```

```
listen "/tmp/unicorn.sock", :backlog => 64
```


- 根据自身情况安装 worker process。
- 在 /tmp/unicorn.sock 中创建 Unix 套接字。

接下来, 编写 Nginx 设置文件。将如下内容保存为 exampleapp.conf 文件。

- dockerbook/Chapter17/exampleapp/exampleapp.conf

> ~/exampleapp/exampleapp.conf

```
upstream unicorn {
    server unix:/tmp/unicorn.sock;
}

server {
    listen 80;
    server_name _;
    root /var/www/exampleapp/public;

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;

        if (!-f $request_filename) {
            proxy_pass http://unicorn;
            break;
        }
    }
}
```

- 将 upstream 项设置为 Unicorn 的 Unix 套接字 /tmp/unicorn.sock。
- 设置 server 项目。
 - 设置 listen 80, 使用 80 号端口。
 - 由于 Nginx 要传送静态文件 (html、js、css 等), 所以设置为 Rails 应用程序目录下的 public 目录。
 - 若进入 Nginx 的请求不是文件名, 则发送到前面设置的 Unix 套接字 (http://unicorn)。这样设置后, Nginx 传送静态问题, 其他 RESTful API 由 Rails 处理。

将如下内容保存为 entrypoint.sh 文件。

- dockerbook/Chapter17/exampleapp/entrypoint.sh

> ~/exampleapp/entrypoint.sh

```
#!/bin/bash

RAILS_ENV=${RAILS_ENV:-"development"}
```

```
bundle exec unicorn -D -c unicorn.rb
nginx
```

- 环境变量 `RAILS_ENV` 若有使用 `docker run` 命令 `-e` 选项设置的值，则使用，否则使用 `development`。
- 向 `unicorn` 使用 `-D` 选项，以后台模式运行，使用 `-c` 选项则将 `unicorn.rb` 文件用作配置文件。
- 由于前面已经在 `nginx.conf` 中设置了 `daemon off;`，故以 `foreground` 方式运行 Nginx 网页服务器。请注意，若不以 `foreground` 方式运行 Nginx，则即使使用 `docker run -d` 生成容器也会立刻终止。

使用 `docker build` 命令创建镜像。

```
~/exampleapp$ sudo docker build --tag rails .
```

16.3 ‣ 编写 PostgreSQL 数据库 Dockerfile 文件

下面创建数据库镜像。创建 `postgresql` 目录后，将如下内容保存为 Dockerfile 文件。

‣ `dockerbook/Chapter17/postgresql/Dockerfile`

```
~$ mkdir postgresql
~$ cd postgresql
```

‣ `~/postgresql/Dockerfile`

```
FROM ubuntu:14.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y postgresql-9.3
```

```
WORKDIR /etc/postgresql/9.3/main
```

```
RUN sed -i "s/#listen_addresses = 'localhost'/listen_addresses = '*'/'g" postgresql.conf
```

```
RUN echo "host all all 0.0.0.0/0 password" >> pg_hba.conf
```

```
EXPOSE 5432
```

```
ADD entrypoint.sh /entrypoint.sh
```

```
RUN chmod +x /entrypoint.sh
```

```
ENTRYPOINT /entrypoint.sh
```

- ▶ 使用 `apt-get update` 将包列表更新为最新状态，然后安装 `postgresql-9.3` 包。
- ▶ 使用 `sed` 修改 `/etc/postgresql/9.3/main` 目录 `postgresql.conf` 文件的内容。将 `#listen_addresses='localhost'` 修改为 `listen_addresses='*'`。若不修改，将无法从外部访问 PostgreSQL。
- ▶ 向 `pg_hba.conf` 文件添加 `host all all 0.0.0.0/0 password`，从外部连接时，将采用密码进行验证。
- ▶ 添加 `entrypoint.sh` 文件，然后设置权限使之可以运行。
- ▶ 设置 `EXPOSE` 为 5432，以连接 5432 端口。
- ▶ 将 `ENTRYPOINT` 设置为 `/entrypoint.sh` 文件，容器启动时，运行该脚本文件。

提示 VOLUME

各位也可以根据自身情况不将数据库存储到容器，而通过设置 `VOLUME` 将其存储到主机。

```
VOLUME ["/var/lib/postgresql"]
```

将如下内容保存为 `entrypoint.sh` 文件。

- ▶ `dockerbook/Chapter17/postgresql/entrypoint.sh`

> ~/postgresql/entrypoint.sh

```
#!/bin/bash

if [ -z $POSTGRES_PASSWORD ]; then
    exit 1
fi

POSTGRES_BIN=/usr/lib/postgresql/9.3/bin/postgres
POSTGRES_CONFIG_FILE=/etc/postgresql/9.3/main/postgresql.conf

POSTGRES_SINGLE="sudo -u postgres $POSTGRES_BIN --single --config-file=$POSTGRES_CONFIG_FILE"
$POSTGRES_SINGLE <<< "ALTER USER postgres PASSWORD '$POSTGRES_PASSWORD';" > /dev/null

exec sudo -u postgres $POSTGRES_BIN --config-file=$POSTGRES_CONFIG_FILE
```

- ▶ 若环境变量中无 `POSTGRES_PASSWORD`，则不运行数据库并退出。
- ▶ 以 `Single` 模式运行 `postgres` 后，设置 `postgres` 账号密码。密码使用环境变量的 `POSTGRES_PASSWORD`。之所以不在 `Dockerfile` 中设置密码而在此处设置密码，是为了使用 `docker run` 命令的 `-e` 选项。使用 `-e` 选项设置的环境变量值只能在 `CMD`、`ENTRYPOINT` 中使用。
- ▶ 运行 `postgres`。与 `Nginx` 相同，以 `foreground` 模式运行 `PostgreSQL`。

使用 `docker build` 命令创建镜像。

```
~/postgresql$ sudo docker build --tag postgresql .
```

16.4 ▶ 创建 Rails 与数据库容器

准备好 Rails 与数据库镜像后，接下来创建容器。首先创建数据库容器。

```
$ sudo docker run -d --name db -e POSTGRES_PASSWORD=examplepassword postgresql
```

- ▶ 创建数据库容器时，使用 `-e` 选项，将 `POSTGRES_PASSWORD` 设置为要使用的 postgres 账号密码。

转到 Rails 应用程序目录后，初始化 Rails 数据库。

```
~$ export POSTGRES_HOST=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" db)
~$ export DB_ENV_POSTGRES_PASSWORD=examplepassword
~$ export RAILS_ENV=development
~$ cd
~$ cd exampleapp
~/exampleapp$ rake db:create
```

- ▶ 使用 `export` 命令，将环境变量的 `POSTGRES_HOST` 设置为 `db` 容器的 IP 地址。
 - » 若在 `docker inspect` 命令中使用 `-f` 选项，则只能输出特定项目。"`{{ .NetworkSettings.IPAddress }}`" 是容器的 IP 地址。
- ▶ 使用 `export` 命令，将环境变量的 `DB_ENV_POSTGRES_PASSWORD` 设置为 PostgreSQL 数据库密码。
- ▶ 使用 `export` 命令，将环境变量的 `RAILS_ENV` 设置为 `development`（根据自身情况，设置为 `production`、`test`）。
- ▶ 运行 `rake db:create`，初始化 Rails 数据库。

提示 使用 MySQL

```

~$ export MYSQL_HOST=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" db)
~$ export DB_ENV_MYSQL_ROOT_PASSWORD=examplepassword
~$ export RAILS_ENV=development
~$ cd
~$ cd exampleapp
~/exampleapp$ rake db:create

```

创建 Rails 容器。

```
$ sudo docker run -d --name example-rails -p 80:80 --link db:db rails
```

► 创建 Rails 容器时，使用 `--link` 选项连接 db 容器与 db 别名。使用 `-p` 选项对外开放 80 号端口，使可以从外部访问。

容器创建完毕后，运行 Web 浏览器，访问服务器的 IP 地址或域名。

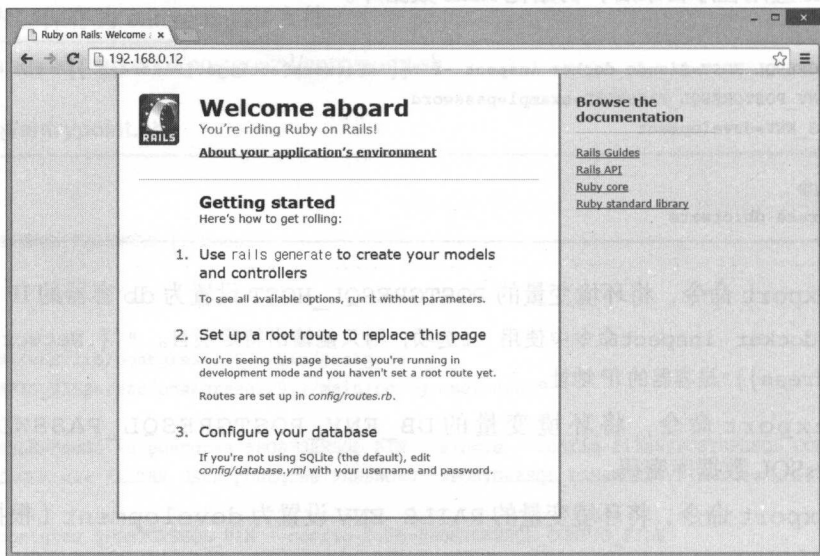


图 16-2 在 Web 浏览器中连接 Rails 容器

第 17 章

DOCKER

使用 Docker 构建 Django 应用

Django 是使用 Python 语言编写的开源 Web 框架。本章将学习使用 Docker 构建 Django 应用的方法。

创建 Docker 镜像前，先搭建 Django 开发环境。

- › 根据 Linux 的发行版本安装 Python pip。
- › 使用 pip 安装 Django。

创建 Django 镜像与数据库镜像。

- › Django 镜像：安装要用作 Web 服务器的 Nginx，并使用 pip 安装 Gunicorn。
- › 数据库镜像：前面使用过 MySQL 与 PostgreSQL 数据库，这次学习使用 Oracle 数据库。安装 MySQL 与 PostgreSQL 的方法请参考 16.2 节与 17.3 节。

创建容器时要使用 `docker run` 命令的 `--link` 选项进行连接，以在 Django 容器中使用数据库容器。

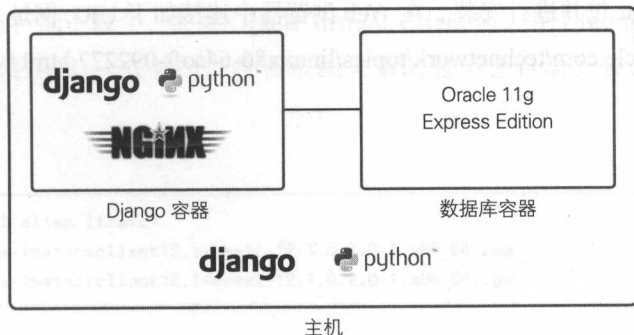


图 17-1 使用 Docker 构建 Django 应用程序

请到我的 GitHub 仓库下载示例文件。

➤ <https://github.com/pyrasis/dockerbook>

17.1 ▶ 安装 Django

创建 Django 镜像前，先搭建 Django 开发环境。根据自身所用的 Linux 发行版本安装 Python pip，然后使用 pip 安装 Django。

运行如下命令，安装 pip 与 Python 开发包。

> Ubuntu

```
$ sudo apt-get install python-pip python-dev
```

> CentOS 6

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ sudo yum install python-pip python-devel gcc
```

> CentOS 7

```
$ sudo yum install http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
$ sudo yum install python-pip python-devel gcc
```

提示 CentOS 7 EPEL 包版本

CentOS 7 EPEL 包版本升级速度快。若无法下载 rpm 文件，请先访问 http://dl.fedoraproject.org/pub/epel/7/x86_64/e/ 检查有无新版本，然后使用 yum 命令安装相应版本。

cx_Oracle 是用于连接并操作 Oracle 数据库的 Python 扩展模块。若要使用，必须从 Oracle 网站下载 Instant Client 包并进行安装。在 Web 浏览器中连接如下 URL 网址。

➤ <http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>

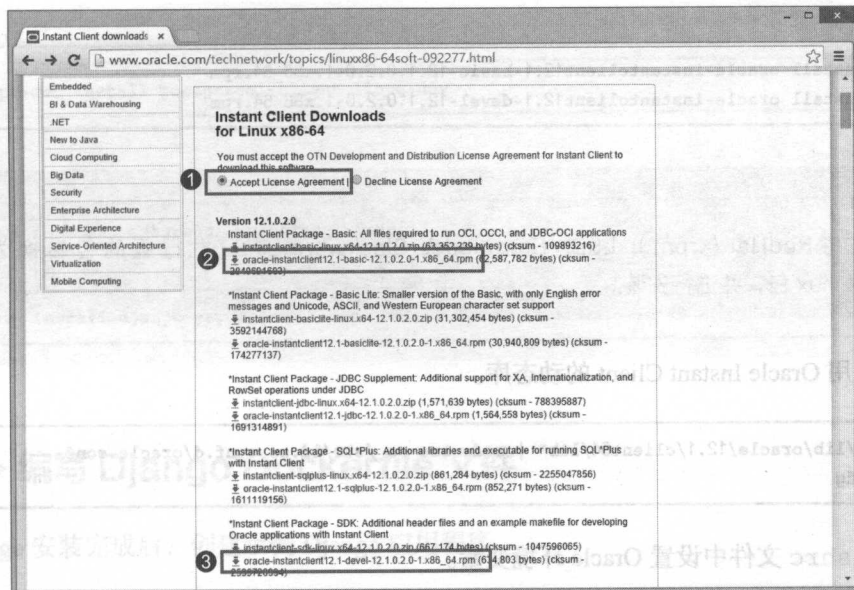


图 17-2 下载 Oracle Instant Client 包

单击 Accept License Agreement 下载如下文件。若想下载文件，必须使用 Oracle 账号登录。若无 Oracle 账号，请先注册申请。

- oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm
- oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm

提示 若想使用 SQL*Plus，则要下载并安装 oracle-instantclient12.1-sqlplus-12.1.0.2.0-1.x86_64.rpm 文件。

下载 rpm 文件后，将其复制到待安装 Django 的 Linux 服务器。在 Windows 中使用 WinSCP (<http://www.winscp.net>) 命令，在 Mac OS X 中使用 sftp 命令即可。

复制文件后，运行如下命令，安装 Oracle Instant Client 包。在 Ubuntu 系统中使用 alien 安装 rpm 包即可。但使用 alien 安装时，存在依赖关系的包不会自动安装，所以需要单独安装 libaio1 包。

> Ubuntu

```
$ sudo apt-get install alien libaio1
$ sudo alien -i oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm
$ sudo alien -i oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm
```

> CentOS

```
$ sudo yum install oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm
$ sudo yum install oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm
```

提示 alien

alien 工具将 RedHat (.rpm)、LSB、Stampede、Slackware Linux 发行版的包转换为 Debian (Ubuntu) Linux 包, 并进行安装。

设置使用 Oracle Instant Client 的动态库。

```
$ echo "/usr/lib/oracle/12.1/client64/lib" | sudo tee -a /etc/ld.so.conf.d/oracle.conf
$ sudo ldconfig
```

在 .bashrc 文件中设置 Oracle 环境变量。

```
~$ echo "export ORACLE_HOME=/usr/lib/oracle/12.1/client64" >> .bashrc
~$ source .bashrc
```

使用 pip 命令安装 django、cx_Oracle 包。此处也使用 env 命令将 ORACLE_HOME 设置到 root 权限。

```
$ sudo env ORACLE_HOME=$ORACLE_HOME pip install django cx_Oracle
```

提示 使用 MySQL、PostgreSQL

若要使用 MySQL, 则安装下列包。

> Ubuntu

```
$ sudo apt-get install mysql-client libmysqlclient-dev
```

> CentOS

```
$ sudo yum install mysql mysql-devel
```

```
$ sudo pip install django MySQL-python
```

若要使用 PostgreSQL, 则安装如下包。

> Ubuntu

```
$ sudo apt-get install postgresql-client libpq-dev
```

> CentOS

```
$ sudo yum install postgresql postgresql-devel
```

```
$ sudo pip install django psycopg2
```

17.2 ▸ 编写 Django Dockerfile 文件

Django 安装完成后，创建示例 Django 应用程序。

```
~$ django-admin.py startproject exampleapp
```

创建 exampleapp 目录后，将前面用于安装 Oracle Instant Client 的 rpm 文件移动到 exampleapp 目录。由于也要在 Django 镜像中安装 Oracle Instant Client，所以需要安装 rpm 文件。

```
$ mv oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm exampleapp/
$ mv oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm exampleapp/
```

打开 exampleapp/exampleapp 目录 settings.py 文件，进行如下修改。

▸ dockerbook/Chapter18/exampleapp/exampleapp/settings.py

> ~/exampleapp/exampleapp/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'XE',
        'USER': 'system',
        'PASSWORD': os.getenv('DB_ENV_ORACLE_PASSWORD'),
        'HOST': os.getenv('ORACLE_HOST') or 'db',
        'PORT': '1521',
    }
}
```

▸ ENGINE：设置为 django.db.backends.oracle，以使用 Oracle。

▸ Name：数据库名称。设置为 XE。

- USER: 设置为 system。
- PASSWORD: 设置使用环境变量的 DB_ENV_ORACLE_PASSWORD。使用 docker run 命令的 --link 选项连接容器时, 所连接的容器的环境变量格式为 < 别名 >_ENV_< 环境变量 >。由于连接容器时将别名指定为 db, 并且数据库容器中环境变量要使用 ORACLE_PASSWORD, 所以最终的完整形式为 DB_ENV_ORACLE_PASSWORD。
- HOST: 使用 or 运算符, 分别设置要在开发环境与数据库容器中使用的数据库主机。开发时, 将环境变量的 ORACLE_HOST 设置为数据库容器的 IP 地址。并且, 由于在连接数据库容器时将别名指定为 db, 故此处设置为 db。
- PORT: 设置为 1512。

提示 使用 MySQL、PostgreSQL

下面是使用 MySQL 的设置。

- dockerbook/Chapter18/exampleapp_mysql/exampleapp/settings.py

> ~/exampleapp/exampleapp/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'exampleapp',
        'USER': 'root',
        'PASSWORD': os.getenv('DB_ENV_MYSQL_ROOT_PASSWORD'),
        'HOST': os.getenv('MYSQL_HOST') or 'db',
        'PORT': '3306',
    }
}
```

下面是使用 PostgreSQL 的设置。

- dockerbook/Chapter18/exampleapp_postgresql/exampleapp/settings.py

> ~/exampleapp/exampleapp/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'exampleapp',
        'USER': 'postgres',
        'PASSWORD': os.getenv('DB_ENV_POSTGRESQL_PASSWORD'),
        'HOST': os.getenv('POSTGRESQL_HOST') or 'db',
        'PORT': '5432',
    }
}
```

将如下内容保存为 Dockerfile 文件。

➤ dockerbook/Chapter18/exampleapp/Dockerfile

> ~/exampleapp/Dockerfile

```
FROM centos:centos7

RUN yum install -y http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
RUN yum install -y python-pip python-devel nginx gcc

RUN echo "daemon off;" >> /etc/nginx/nginx.conf
ADD exampleapp.conf /etc/nginx/conf.d/exampleapp.conf

WORKDIR /tmp
ADD oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm /tmp/
ADD oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm /tmp/
RUN yum install -y oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm
RUN yum install -y oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm
RUN rm -rf *.rpm

ENV ORACLE_HOME /usr/lib/oracle/12.1/client64
RUN echo "/usr/lib/oracle/12.1/client64/lib" > /etc/ld.so.conf.d/oracle.conf && ldconfig
RUN pip install django gunicorn cx_Oracle

ADD ./ /var/www/exampleapp
WORKDIR /var/www/exampleapp
RUN chmod +x entrypoint.sh
RUN rm -rf *.rpm

EXPOSE 80

ENTRYPOINT ./entrypoint.sh
```

之前都以 Ubuntu 14.04 为基础创建镜像。下面以 CentOS 7 为基础创建镜像。

- 使用 FROM 命令指定以 centos:centos7 为基础创建镜像。
- 使用 yum install 命令安装 EPEL 包，也安装 python-pip、python-devel、nginx、gcc。
- 设置以 foreground（非守护进程方式）方式运行 nginx。请注意，若以守护进程方式运行 nginx，Docker 容器会直接停止。
- 向 /etc/nginx/conf.d 目录添加 exampleapp.conf 文件。
- 向 /tmp 目录添加 Oracle Install Client rpm 文件后，使用 yum install 命令进行安装。安装完毕后，删除 rpm 文件。
- 使用 ENV 命令设置 ORACLE_HOME 环境变量。/usr/lib/oracle/12.1/client64 是 Oracle Install Client 的安装路径。

- 在 `/etc/ld.so.conf.d` 目录中创建 `oracle.conf` 文件，运行 `ldconfig` 命令，使 Oracle Instant Client 的动态库可用。
- 使用 `pip install` 命令安装 `django`、`gunicorn`、`cx_Oracle`。
- 将 Django 应用程序目录添加到 `/var/www/exampleapp` 目录。
- 设置可运行 `entrypoint.sh` 文件的权限。
- 将 `EXPOSE` 设置为 80，使可以连接到 80 号端口。
- 将 `ENTRYPOINT` 设置为 `./entrypoint.sh` 文件，容器启动时运行该脚本文件。

添加 Django 应用程序目录（`exampleapp`）前，事先要将 `oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm`、`oracle-instantclient12.1-devel-12.1.0.2.0-1.x86_64.rpm` 文件添加到 `/tmp` 目录并安装。若添加 `exampleapp` 目录后再安装 `rpm` 文件，则每当 `exampleapp` 目录的内容发生变化时都会安装 `rpm` 文件，所以镜像创建时间会很长。更详细的内容请参考 17.2 节。

提示 使用 MySQL、PostgreSQL

下面是使用 MySQL 的设置。

- `dockerbook/Chapter18/exampleapp_mysql/Dockerfile`

> `~/exampleapp/Dockerfile`

```
RUN yum install -y python-pip python-devel nginx gcc mysql-devel
```

```
RUN pip install django gunicorn MySQL-python
```

下面是使用 PostgreSQL 的设置。

- `dockerbook/Chapter18/exampleapp_postgresql/Dockerfile`

> `~/exampleapp/Dockerfile`

```
RUN yum install -y python-pip python-devel nginx gcc postgresql-devel
```

```
RUN pip install django gunicorn psycopg2
```

接下来，编写 Nginx 配置文件。将如下内容保存为 `exampleapp.conf` 文件。

- `dockerbook/Chapter18/exampleapp/exampleapp.conf`

> `~/exampleapp/exampleapp.conf`

```
upstream gunicorn {
    server unix:/tmp/gunicorn.sock;
}
```

```
server {
    listen 80;
    server_name _;

    location /static/ {
        alias /var/www/exampleapp/static/;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;

        if (!-f $request_filename) {
            proxy_pass http://gunicorn;
            break;
        }
    }
}
```

- 将 upstream 设置为 Gunicorn 的 Unix 套接字 /tmp/gunicorn.sock。
- 设置 server 项目。
 - » 设置 listen 80，使用 80 号端口。
 - » Nginx 会传送静态文件（html、js、css 等），所以设置为 Django 应用程序目录下的 static 目录（请根据自身情况进行修改）。
 - » 若进入 Nginx 的请求不是文件名，则发送到前面设置的 Unix 套接字（http://gunicorn）。这样设置后，静态文件由 Nginx 传送，其他 RESTful API 由 Django 处理。

将如下内容保存为 entrypoint.sh 文件。

- dockerbook/Chapter18/exampleapp/entrypoint.sh

> ~/exampleapp/entrypoint.sh

```
#!/bin/bash

gunicorn exampleapp.wsgi -b unix:/tmp/gunicorn.sock -D
nginx
```

- 将 gunicorn 设置为 exampleapp.wsgi，格式为 <Django 应用程序名>.wsgi。
 - » 使用 -b 选项，在 /tmp/gunicorn.sock 创建 Unix 套接字。
 - » 使用 -D 选项，以守护进程模式运行。
- 由于前面在 nginx.conf 中设置了 daemon off;，所以 Nginx Web 服务器以 foreground 方式运行。请注意，若不以 foreground 方式运行 Nginx，则即使使用 docker run -d 创

建也会立刻终止。

使用 docker build 命令创建镜像。

```
~/exampleapp$ sudo docker build --tag django .
```

17.3 ▸ 编写 Oracle 数据库 Dockerfile 文件

创建数据库镜像前，先要从 Oracle 网站下载 Oracle 11g Express Edition 包。在 Web 浏览器中打开如下 URL 地址。

▸ <http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html>

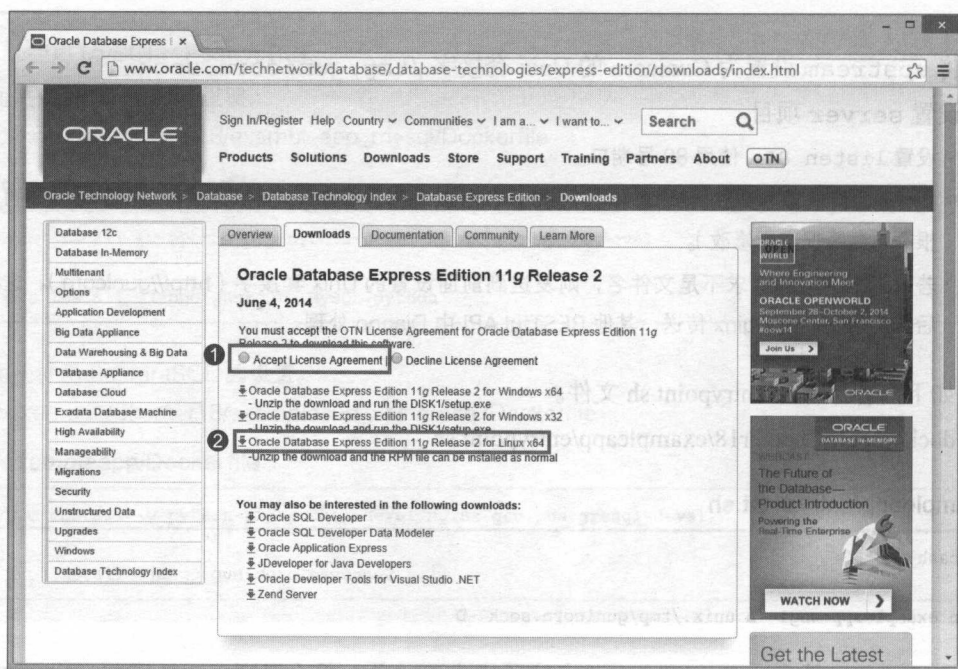


图 17-3 下载 Oracle 11g Express Edition 包

单击 Accept License Agreement，下载 Oracle Database Express Edition 11g Release 2 for Linux x64。若要下载文件，必须先使用 Oracle 账号登录。若无 Oracle 账号，请先注册申请。

将 oracle-xe-11.2.0-1.0.x86_64.rpm.zip 文件解压缩后，在 Disk1 目录中可以看到 rpm 文件。将该 rpm 文件复制到要创建数据库镜像的 Linux 服务器。在 Windows 系统中使用 WinSCP

(<http://www.winscp.net>), 在 Mac OS X 中使用 `sftp` 命令即可。

接下来, 创建数据库镜像。创建 `oracle` 目录后, 将如下内容保存为 `Dockerfile` 文件。

► `dockerbook/Chapter18/oracle/Dockerfile`

```
~$ mkdir oracle
~$ cd oracle
```

► `~/oracle/Dockerfile`

```
FROM centos:centos7

RUN yum install -y net-tools bc openssh-server
RUN rm -rf /var/lock
RUN mkdir -p /var/lock/subsys

WORKDIR /tmp
ADD oracle-xe-11.2.0-1.0.x86_64.rpm /tmp/
RUN yum install -y oracle-xe-11.2.0-1.0.x86_64.rpm
RUN rm -rf oracle-xe-11.2.0-1.0.x86_64.rpm

WORKDIR /u01/app/oracle/product/11.2.0/xe/config/scripts
RUN sed -i "s/memory_target/#memory_target/g" init.ora
RUN sed -i "s/memory_target/#memory_target/g" initXETemp.ora
RUN echo -e "pga_aggregate_target=200540160\nsga_target=601620480" >> init.ora
RUN echo -e "pga_aggregate_target=200540160\nsga_target=601620480" >> initXETemp.ora

RUN sed -i "s/#PermitRootLogin/PermitRootLogin/g" /etc/ssh/sshd_config
RUN sshd-keygen

ADD entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

EXPOSE 22
EXPOSE 1521
EXPOSE 8080

ENTRYPOINT /entrypoint.sh
```

- 使用 `yum install` 命令安装 `net-tools`、`bc`、`openssh-server` 包。
- 删除 `/var/lock` 文件, 创建 `/var/lock/subsys` 目录。
- 将 `oracle-xe-11.2.0-1.0.x86_64.rpm` 文件添加到 `/tmp` 目录, 使用 `yum install` 命令安装。安装完成后, 删除 `rpm` 文件。
- 修改 Oracle 配置文件 `init.ora`、`initXETemp.ora`。
 - » 在 `memory_target` 前添加 `#`, 不使用。
 - » 添加 `pga_aggregate_target=200540160`、`sga_target=601620480`。

- 设置 SSH 服务器。
 - » 删除 /etc/ssh/sshd_config 的 PermitRootLogin 部分中的 #，允许以 root 账号登录。
 - » 使用 sshd-keygen 命令创建服务器密钥对。
- 添加 entrypoint.sh 文件后，设置权限使其可以运行。
- 将 EXPOSE 设置为 22、1521、8080，使可以连接到 22、1521、8080 端口。
- 将 ENTRYPOINT 设置为 /entrypoint.sh 文件，容器启动时运行该脚本文件。

将如下内容保存为 entrypoint.sh 文件。

➤ dockerbook/Chapter18/oracle/entrypoint.sh

> ~/oracle/entrypoint.sh

```
#!/bin/bash

if [ -z $ORACLE_PASSWORD ]; then
    exit 1
fi

echo "export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe" > /etc/profile.d/oracle.sh
echo "export ORACLE_SID=XE" >> /etc/profile.d/oracle.sh
echo "export PATH=$PATH:/u01/app/oracle/product/11.2.0/xe/bin" >> /etc/profile.d/oracle.sh
source /etc/profile.d/oracle.sh

printf 8080\\n1521\\n$ORACLE_PASSWORD\\n$ORACLE_PASSWORD\\nn | /etc/init.d/oracle-xe configure
echo "root:$ORACLE_PASSWORD" | chpasswd

/usr/sbin/sshd -D
```

- 若环境变量中不存在 ORACLE_PASSWORD，则不运行数据库并退出。
- 在 /etc/profile.d/oracle.sh 文件中设置 ORACLE_HOME、ORACLE_SID、PATH 环境变量，使用 source 命令应用设置。
 - » /u01/app/oracle/product/11.2.0/xe 是 Oracle 数据库的安装目录。
 - » 由于是 Oracle 11g Express Edition，故安装 XE。
 - » 设置可执行文件路径，以使用 sqlplus 等命令。
- 使用 printf 命令，运行 /etc/init.d/oracle-xe configure 时加入设置值。设置为 8080、1521，密码设置为环境变量的 ORACLE_PASSWORD 值。
 - » 运行 /etc/init.d/oracle-xe configure，自动运行 Oracle 服务。
- 将 root 账号的密码修改为环境变量的 ORACLE_PASSWORD 值，以使用 SSH 进行连接。
- 运行 /usr/sbin/sshd -D 命令，以 foreground 方式运行 SSH 服务器。

使用 docker build 命令创建镜像。

```
~/oracle$ sudo docker build --tag oracle .
```

17.4 ▸ 创建 Django 与数据库容器

Django 与数据库镜像准备完毕后，开始创建容器。首先创建数据库容器。

```
$ sudo docker run -d --name db -e ORACLE_PASSWORD=examplepassword oracle
```

- 创建数据库容器时，使用 `-e` 选项，将 `ORACLE_PASSWORD` 设置为要使用的 `system` 账号的密码。

转到 Django 应用程序目录，然后初始化 Django 数据库，设置管理员账号。

```
~$ export ORACLE_HOST=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" db)
```

```
~$ export DB_ENV_ORACLE_PASSWORD=examplepassword
```

```
~$ cd
```

```
~$ cd exampleapp
```

```
~/exampleapp$ ./manage.py syncdb
```

Operations to perform:

Apply all migrations: admin, contenttypes, auth, sessions

Running migrations:

Applying contenttypes.0001_initial... OK

Applying auth.0001_initial... OK

Applying admin.0001_initial... OK

Applying sessions.0001_initial... OK

You have installed Django's auth system, and don't have any superusers defined.

Would you like to create one now? (yes/no): **yes**

Username (leave blank to use 'pyrasis'): **admin**

Email address: **admin@example.com**

Password: <输入密码>

Password (again): <输入密码>

Superuser created successfully.

- 使用 `export` 命令，将环境变量的 `ORACLE_HOST` 设置为 `db` 容器的 IP 地址。
 - 若在 `docker inspect` 命令中使用 `-f` 选项，则只能输出特定项目。"`{{ .NetworkSettings.IPAddress }}`" 是容器的 IP 地址。
- 使用 `export` 命令，将环境变量的 `DB_ENV_POSTGRESQL_PASSWORD` 设置为 Oracle 数据库的密码。
- 运行 `manage.py syncdb`，初始化 Django 数据库。若出现管理员（`superuser`）设置，则输入 `yes` 并设置管理员账号、电子邮件、密码。

提示 使用 MySQL、PostgreSQL

下面是使用 MySQL 的设置。

```
~$ export MYSQL_HOST=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" db)
~$ export DB_ENV_MYSQL_ROOT_PASSWORD=examplepassword
~$ mysql -h $MYSQL_HOST -uroot -p$DB_ENV_MYSQLROOT_PASSWORD \
  -e "create database exampleapp"
~$ cd
~$ cd exampleapp
~/exampleapp$ ./manage.py syncdb
```

下面是使用 PostgreSQL 的设置。

```
~$ export POSTGRES_HOST=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" db)
~$ export DB_ENV_POSTGRES_PASSWORD=examplepassword
~$ PGPASSWORD=$DB_ENV_POSTGRES_PASSWORD psql -h $POSTGRES_HOST -U postgres \
  -c "CREATE DATABASE exampleapp WITH ENCODING 'UTF8' TEMPLATE template0"
~$ cd
~$ cd exampleapp
~/exampleapp$ ./manage.py syncdb
```

创建 Django 容器。

```
$ sudo docker run -d --name example-django --link db:db -p 80:80 django
```

- 创建 Django 容器时，使用 `--link` 选项连接 `db` 容器与 `db` 别名。使用 `-p` 选项，使可以从外部访问 80 号端口。

容器创建完毕后，运行 Web 浏览器，访问服务器的 IP 地址或域名。

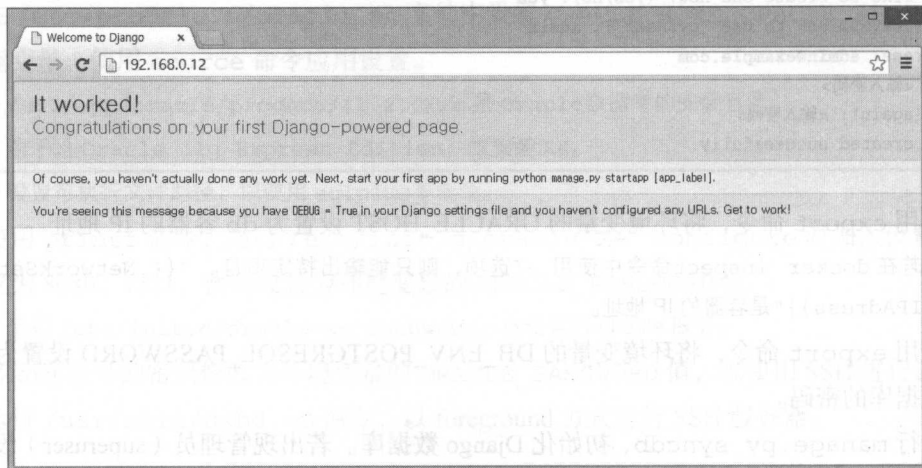


图 17-4 在 Web 服务器中访问 Django 容器

第 18 章

DOCK ER

Docker 应用案例

到目前为止，我们已经学习了 Docker 的基本用法以及使用其构建 Web 应用的方法。本章将向各位介绍 4 种典型的 Docker 应用案例。

18.1 与负载均衡相关的自动伸缩

随着云服务的出现，人们现在只需点击几次即可轻松使用数十、数百台服务器。因此，现在不再像从前那样 1 年 365 天开着服务器，而可以根据需要随时创建服务器实例进行使用。尤其是灵活使用负载均衡（用于分担负荷）与自动伸缩（Auto Scaling）功能，能够充分满足急剧增长的大量通信需求。

运用 Docker 将其与云服务中提供的功能结合，构建更便利的自动伸缩环境。

如图 18-1 所示，在 AWS 的 ELB 负载均衡与自动伸缩中使用 Docker。

- ▶ ELB 负载均衡将来自外部的通信流量分配给自动伸缩组中的 EC2 实例。
- ▶ 自动伸缩依据流量创建或删除 EC2 实例。
- ▶ 新创建的 EC2 实例从 Docker 注册表获取镜像，然后以容器运行。
- ▶ 与使用安装了所有服务环境的 AMI（Amazon Machine Images）相比，创建 EC2 实例时使用 Docker 会更方便。使用 Docker 可以使 EC2 实例与内部测试环境以及开发人员的 PC 拥有一致的环境。
- ▶ AMI 中只使用安装 Docker 的状态。
- ▶ 在 AWS 内部构建 Docker 注册表，以快速下载 Docker 镜像。
- ▶ 使用 CoreOS 能够更方便地部署 Docker 镜像，以及构建高可用性的服务。

借助这种方式也可以在其他云服务中轻松构建一致的应用。

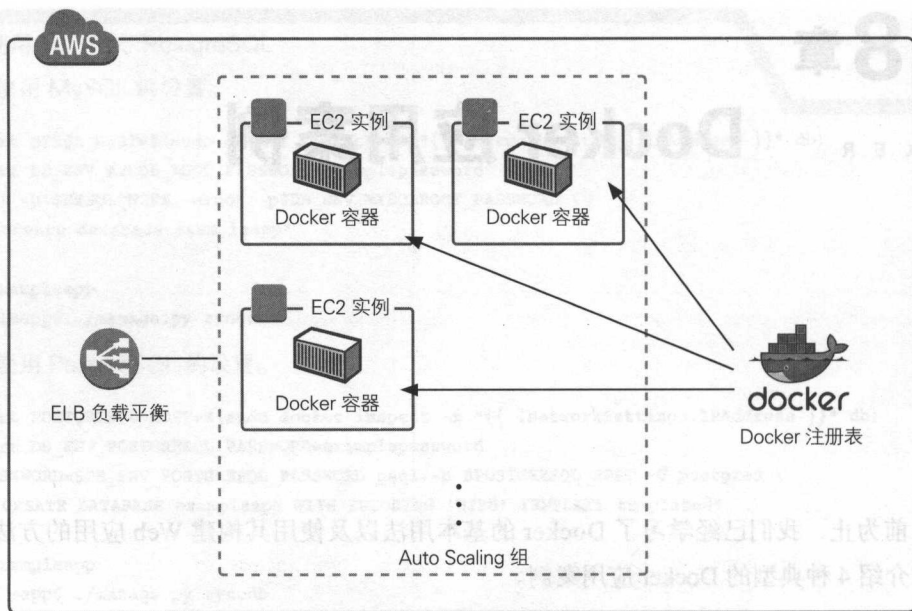


图 18-1 在 AWS 中使用 Docker

18.2 ▶ 整合开发、测试、运营

不久前出现了 DevOps (Dev+Ops) 这个新词汇，这是由 Chef 的开发公司 Opscode 创造的。开发 (Development) 与运营 (Operation) 组织通常彼此分离，独立开展业务，导致业务效率低下，沟通成本增加。

DevOps 将开发、测试、运营整个过程全部自动化，缩短发布周期，使用标准化工具大大降低沟通成本。

如图 18-2 所示，使用 Docker 整合开发、测试、运营。

- ▶ 开发人员将源代码上传到源代码服务器 / 构建服务器后，会自动创建 Docker 镜像。
 - ▶▶ 源代码服务器由 Git、Subversion、Mercurial、Perforce 等工具组成。
 - ▶▶ 构建服务器由 Jenkins、CruiseControl 等工具组成。
- ▶ 自动创建的 Docker 镜像会被使用事先定义的测试用例自动测试，或者直接用人工测试。
 - ▶▶ 对于网站，可以使用 phantomjs 等 Headless Web 浏览器进行测试。
- ▶ 测试完毕后，将 Docker 镜像部署到提供服务的服务器，并以容器运行。

开发产品或服务时，测试人员一直说功能不正常，服务也时常因为一些细小的问题而发生故障。此时，开发人员常说的一句话就是：“在我的电脑上一切正常啊！”。

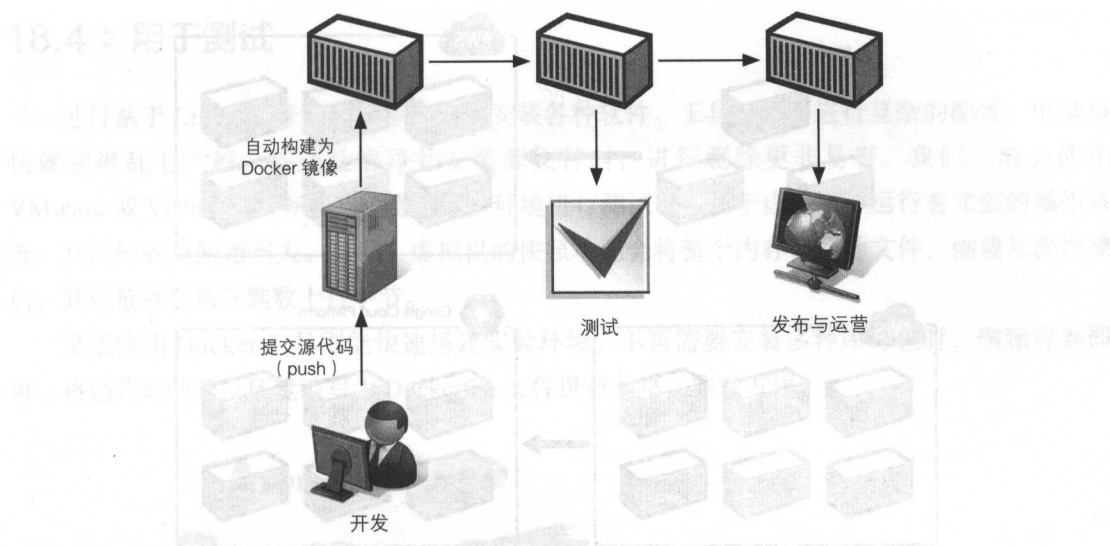


图 18-2 使用 Docker 整合开发、测试、运营

借助 Docker 的帮助，开发人员在自身的 PC 中创建镜像与容器，在与服务器环境一致的状态下进行开发，而测试人员则可以在与服务器环境、开发人员 PC 一致的状态下进行测试。也就是说，通过 Docker 可以避免开发人员 PC 环境、测试环境、服务环境不一致引发的各种问题。

随着 DevOps 技术的发展，也许未来某一天将不再需要运营部门，最终形成一个 NoOps 环境。

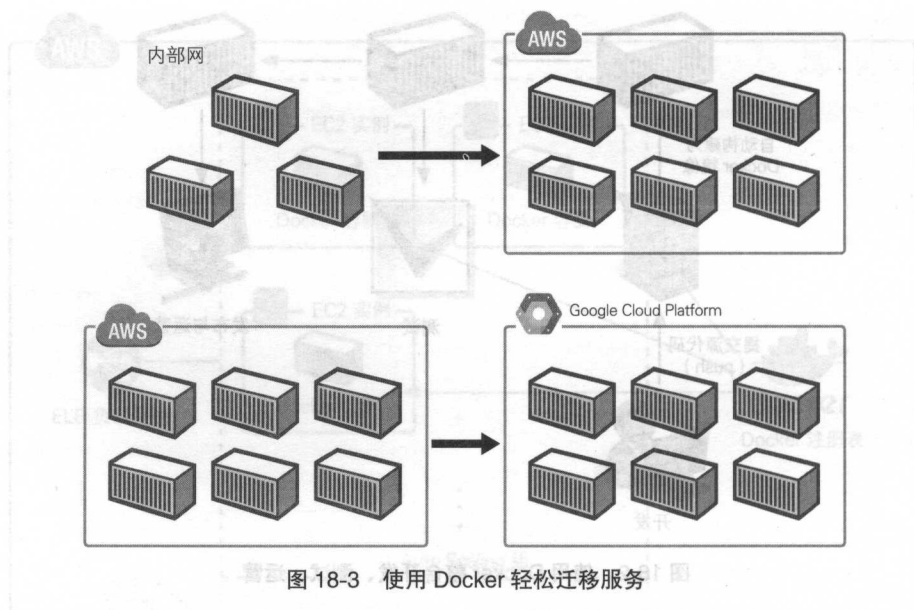
18.3 轻松迁移服务

只要安装 Docker，无论何地都可以创建 Docker 容器，所以可以轻松实现从内部网向云服务，或者从一个云服务向另一个云服务的迁移。

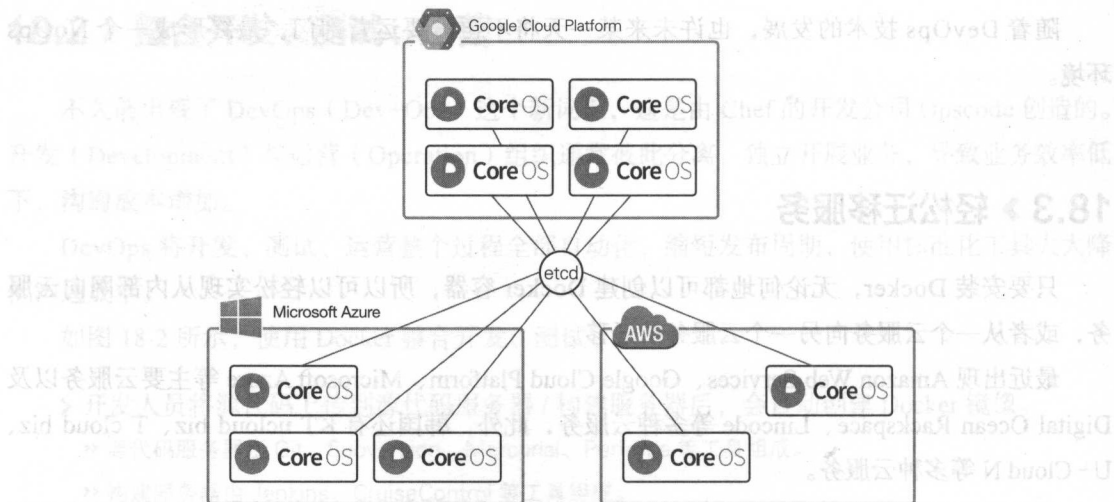
最近出现 Amazon Web Services、Google Cloud Platform、Microsoft Azure 等主要云服务以及 Digital Ocean Rackspace、Lincode 等多种云服务，此外，韩国还有 KT ucloud biz、T cloud biz、U+ Cloud N 等多种云服务。

随着云服务竞争越来越激烈，提供云服务的新企业不断涌现，云服务价格正变得越来越便宜，云服务也越来越稳定。以前，某项服务完成构建后就难以迁移；而进入云服务时代后，这类问题出现明显改观。借助 Docker 工具构建服务后，可以轻松将其迁移到价格低廉的云服务。

图 18-3 简单展现了使用 Docker 工具进行服务迁移的情形。只要拥有 Docker 镜像，无论何地都能提供服务，所以能够轻松实现从公司内部网向云服务的迁移，或者从 Amazon Web Service 向 Google Cloud Platform 迁移。



尤其是灵活使用 Docker 这一共同接口，用户也可以将一部分服务构建到 Amazon Web Service，而将另一部分服务构建到 Microsoft Azure 等多种云平台，发布跨云平台的服务。



事实上，CoreOS 系统用于在多种云服务中组建集群，如图 18-4 所示。

18.4 用于测试

进行基于 Linux 的开发时，常常需要安装各种软件、工具，还要进行复杂的配置，电脑很快就变得乱七八糟。特别是编译并安装源文件时，进行删除更非易事。我们一般会使用 VMware 或 VirtualBox 等虚拟机搭建开发环境进行测试，但由于虚拟机中运行着完整的操作系统，其占据的空间相当大。另外，虚拟机的快照功能会将整个内存保存为文件，创建几次快照后，其容量就会飙升到数十吉字节。

灵活使用 Docker 工具则能快速搭建实验环境。不再需要安装多种库与包时，删除容器即可。将创建好的开发环境编写为 Dockerfile 文件进行共享，非常方便。

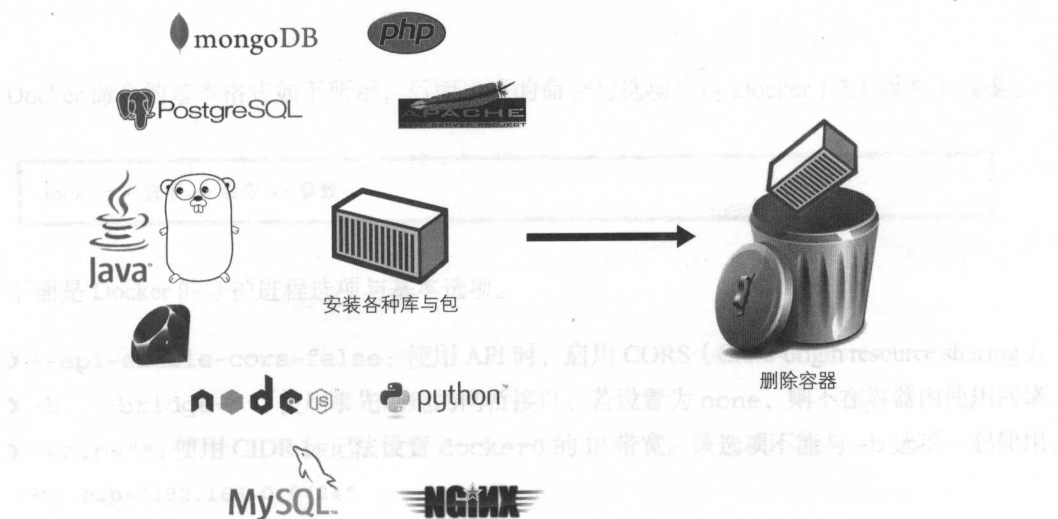


图 18-5 灵活使用 Docker 工具进行测试

若要进行简单的测试，可以使用 `docker run` 命令的 `--rm` 选项，非常方便。如下所示，使用 `ubuntu:14.04` 镜像完成测试后，使用 `exit` 命令退出即可删除容器。

```
$ sudo docker run -i -t --rm ubuntu:14.04 /bin/bash
root@e106f864897e:/# apt-get update
root@e106f864897e:/# apt-get install nodejs
root@e106f864897e:/# exit
```


第 19 章

DOCKER

Docker 命令与选项列表

Docker 命令的基本格式如下所示，后面出现的命令与选项均以 Docker 1.3.1 版本为基准。

```
docker < 选项 >< 命令 >< 参数 >
```

下面是 Docker 的守护进程选项与基本选项。

- > `--api-enable-cors=false`: 使用 API 时，启用 CORS (Cross-origin resource sharing)。
- > `-b`、`--bridge=""`: 使用事先创建的网桥接口。若设置为 `none`，则不在容器内使用网络。
- > `--bip=""`: 使用 CIDR 标记法设置 `docker0` 的 IP 带宽。该选项不能与 `-b` 选项一起使用。
 - » `--bip="192.168.0.0/24"`
- > `-D`、`--debug=false`: 启用调试模式。
- > `-d`、`--daemon=false`: 以守护进程模式运行。
- > `--dns=[]`: 设置 Docker 要使用的 DNS 服务器。
- > `--dns-search=[]`: 设置 Docker 要使用的 DNS 搜索域。若设置为 `example.com`，则向 DNS 服务器查询 `hello` 时，将首先查找 `hello.example.com`。
- > `-e`、`--exec-driver="native"`: 设置 Docker 运行驱动，可以设置为 `native` 与 `lxc`。
- > `--fixed-cidr=""`: 固定分配 IPv4 地址的带宽。该 IP 地址必须在 `-b` 选项设置的网桥网络或 `--bip` 设置的 IP 网段内。
 - » `--fixed-cidr="172.17.42.0/29"`
 - » 若设置为 `172.17.42.0/29`，则在 `172.17.42.0~172.17.42.7` 分配 IP 地址。
- > `-G`、`--group="docker"`: 以守护进程模式运行时，使用 `-H` 选项创建 Unix 套接字后，设置该 Unix 套接字所在的组。使用 `""` 空字符串表示不设置分组。

- > -g、--graph="/var/lib/docker": 设置 Docker 使用目录的顶层路径。
- > -H、--host []: 以守护进程模式运行时, 设置套接字路径。更详细的内容请参考第 14 章。
 - >> tcp://<IP 地址或域名>:<端口号>
 - >> unix:///<套接字路径>
 - >> fd://* 或 fd://socketfd
- > --icc=true: 开启容器间通信。
- > --insecure-registry=[]: 使用私有证书搭建 Docker 注册服务器时, 设置 Docker 注册服务器的域名。详细内容请参考 6.1.4 节。
- > --ip=0.0.0.0: 使用 docker run 命令的 -p 选项将端口暴露在外时, 设置要绑定的默认 IP 地址。
- > --ip-forward=true: 开启 net.ipv4.ip_forward。
- > --ip-masq=true: 为网桥上的 IP 地址开启 IP 伪装 (masquerading)。
- > --iptables=true: 开启 iptables 规则。
- > --mtu=0: 设置容器的网络最大传输单元 (MTU, Maximum transmission unit)。若不设置, 则使用路由器的默认 MTU 或设置为 1500。
- > -p、--pidfile="/var/run/docker.pid": 设置 PID 文件路径。
- > --registry-mirror=[]: 设置 Docker Registry 的镜像地址。
- > -s、--storage-driver="": 设置存储驱动, 默认为 aufs, 也可以设置为 devicemapper。
- > --selinux-enabled=false: 开启 SELinux。SELinux 尚不支持 BTRFS 存储驱动。
- > --storage-opt=[]: 设置存储驱动选项。
- > --tls=false: 使用 TLS。更详细的内容请参考 14.2 节。
- > --tlscacert="/home/exampleuser/.docker/ca.pem": 设置要在远程证书中使用的 CA 证书文件的路径。
- > --tlscert="/home/exampleuser/.docker/cert.pem": 设置证书文件路径。
- > --tlskey="/home/exampleuser/.docker/key.pem": 设置密钥文件路径。
- > --tlsverify=false: 使用 TLS 远程证书, 守护进程与客户端全部使用证书验证。
- > -v、--version=false: 打印版本信息。

19.1 > attach

attach 命令用于将标准输入 (stdin) 与标准输出 (stdout) 连接到正在运行的容器。

```
docker attach < 选项 >< 容器名称, ID>
```

- ▶ `--no-stdin=false`: 不连接标准输入。
- ▶ `--sig-proxy=true`: 将所有信号传递给进程（非 TTY 模式时也一样），但不传送 SIGCHLD、SIGKILL、SIGSTOP 信号。经常使用的信号如下所示：
 - » SIGINT: Interrupt 信号，输入 Ctrl+C 时发生。
 - » SIGQUIT: Quit 信号，输入 Ctrl+\ 时发生。
 - » EOF: 终止 attach 状态，输入 Ctrl+D 时发生。

提示 SIGCHLD、SIGKILL、SIGSTOP

- SIGCHLD: 子进程暂停或终止时，向父进程传递该信号。
- SIGKILL: 该信号用于强制终止进程。
- SIGSTOP: 该信号用于暂停进程。

一般先会运行 Bash 等 shell，然后使用 `docker attach` 命令连接到容器，再运行各种命令。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker attach hello
root@de6e3b886fb1:/#
```

也可以使用 `docker attach` 命令查看进程的输出内容。

```
$ sudo docker run -d --name hello ubuntu:14.04 \
  /bin/bash -c "while true; do echo Hello World; sleep 1; done"
$ sudo docker attach hello
Hello World
Hello World
Hello World
```

19.2 ▶ build

`build` 命令使用 Dockerfile 文件创建镜像。

```
docker build < 选项 >< Dockerfile 路径 >
```

Dockerfile 路径可以是本地路径，也可以是 URL 路径。若设置为 -，则从标准输入获取 Dockerfile 的内容。

- --force-rm=false: 镜像创建失败时，删除临时容器。
 - --no-cache=false: 不使用之前构建中创建的缓存。为了缩短镜像创建时间，Docker 会对 Dockerfile 的各个过程进行缓存，使用该选项将不使用缓存而重新创建镜像。
 - -q、--quiet=false: 不显示 Dockerfile 的 RUN 运行的输出结果。
 - --rm=true: 镜像创建成功时，删除临时容器。
 - -t、--tag="": 设置注册名称、镜像名称、标签。格式为 <注册名称>/<镜像名称>:<标签>。
- ```

» hello
» hello:0.1
» exampleuser/hello
» exampleuser/hello:0.1

```

一般会在 Dockerfile 文件所在的路径下运行 `docker build` 命令。也可以设置 Dockerfile 文件所在的特定路径。

---

```

$ sudo docker build -t hello .
$ sudo docker build -t hello /opt/hello
$ sudo docker build -t hello ../..

```

---

下面使用网络上的 Dockerfile 的 URL 创建镜像。

---

```

$ sudo docker build -t hello https://raw.githubusercontent.com/kstaken/dockerfile-examples/master/apache/Dockerfile

```

---

下面将 Dockerfile 路径设置为 -，从标准输入获取 Dockerfile 文件内容。

---

```

$ echo -e "FROM ubuntu:14.04\nRUN apt-get update" | sudo docker build -t hello -
$ cat Dockerfile | sudo docker build -t hello -
$ sudo docker build -t hello - < Dockerfile

```

---

既可以使用 `echo` 命令直接输出字符串，也可以使用 `cat` 命令将文件内容发送至管道。

## 19.3 ▶ Commit

commit 命令用于从容器的修改项创建新的镜像。

```
docker commit < 选项 > < 容器名称, ID > < 注册名称 > / < 镜像名称 > : < 标签 >
```

- ▶ -a, --author="": 设置镜像创建者的有关信息, 比如 "Foo Bar<foo@bar.com>"。
- ▶ -m, --message="": 设置有关变更事项的日志信息。
- ▶ -p, --pause=true: 创建镜像时暂停容器。

---

```
$ sudo docker run -i -t --name hello ubuntu:14.04 /bin/bash
root@3425069b321a:/# echo "Hello World" > hello.txt
root@3425069b321a:/# exit
$ sudo docker commit -a "Foo Bar <foo@bar.com>" -m "add hello.txt" hello hello:0.2
```

---

## 19.4 ▶ cp

cp 命令用于将容器的目录或文件复制到主机。若将 cp 命令中的路径设置为目录, 则将该目录下的所有内容复制到主机。

```
docker cp < 容器名称 > : < 路径 > < 主机路径 >
```

---

```
$ sudo docker run -i -t --name hello ubuntu:14.04 /bin/bash
root@3425069b321a:/# echo "Hello World" > hello.txt
root@3425069b321a:/# exit
$ sudo docker cp hello:/hello.txt .
$ ls
hello.txt
```

---

使用如下命令将容器的整个 /etc 目录复制到主机。

---

```
$ sudo docker cp hello:/etc .
```

---

## 19.5 create

create 命令使用指定的镜像创建容器。与 run 命令不同，使用 create 命令只能创建容器而并不启动。

```
docker create < 选项 > < 镜像名称 , ID > < 命令 > < 参数 >
```

设置选项值时，= 与 " 也可以省略。

➤ -a, --attach=[] : 将标准输入、标准输出、标准错误连接到容器。

    » --attach="stdin"

➤ --add-host=[] : 向容器的 /etc/hosts 添加主机名与 IP 地址。

    » --add-host=hello:192.168.0.10

➤ -c, --cpu-shares=0 : 设置 CPU 资源分配。默认设置值为 1024，各值为相对值。

    » 若设置为 --cpu-shares=2048，则分配默认值 2 倍的 CPU 资源。

    » 在 Linux 内核的 cgroups 中使用该设置值。

➤ --cap-add[] : 设置容器中使用的 cgroups 的特定 Capability。若设置为 ALL，则使用所有 Capability。

    » 像 --cap-add="MKNOD" --cap-add="NET\_ADMIN" 一样进行设置。关于所有 Capability 列表请参考如下链接。

<http://linux.die.net/man/7/capabilities>

➤ --cap-drop=[] : 从容器删除 cgroups 的特定 Capability。

➤ --cidfile="" : 设置 cid 文件路径。cid 中存储着所创建容器的 ID。

➤ --cpuset="" : 在多核 CPU 中设置要运行容器的核心。

    » 若设置为 --cpuset="0,1"，则使用第一个与第二个 CPU 内核。

    » 若设置为 --cpuset="0-3"，则使用从第一到第三个 CPU 内核。

➤ --device=[] : 添加主机设备到容器，格式为 < 主机设备 > : < 容器设备 >。

    » 若设置为 --device="/dev/sda1:/dev/sda1"，则在容器也可以使用主机的 /dev/sda1 块设备。

➤ --dns=[] : 设置容器要使用的 DNS 服务器。

    » --dns="8.8.8.8"

➤ --dns-search=[] : 设置容器要使用的 DNS 搜索域。

    » 若设置为 --dns-search="example.com"，则向 DNS 查询 hello 时，优先查找 hello.example.com。

➤ -e, --env=[] : 向容器设置环境变量。一般用于传递设置值或密码。



- » -e MYSQL\_ROOT\_PASSWORD=examplepassword
- > --entrypoint="": 忽略 Dockerfile 的 ENTRYPOINT 设置, 强制设置为其他值。
  - » --entrypoint="/bin/bash"
- > --env-file=[]: 向容器应用设有环境变量的文件。
  - » --env-file="/etc/environment"
- > --expose=[]: 仅连接容器的端口与主机, 并不暴露在外。
  - » --expose="3306"
- > -h, --hostname="": 设置容器的主机名。
- > -i, --interactive=false: 激活标准输入, 即使未与容器连接 (attach), 也维持标准输入。一般使用该选项向 Bash 输入命令。
- > --link=[]: 进行容器间连接, 格式为 < 容器名称 >:< 别名 >。
  - » --link="db:db"
- > --lxc-conf=[]: 若使用 LXC 驱动, 则可以设置 LXC 选项。
  - » --lxc-conf="lxc.cgroup.cpuset.cpus = 0,1"
- > -m, --memory="": 设置内存限制, 格式为 < 数字 >< 单位 >, 单位可以使用 b、k、m、g。
  - » --memory="100000b"
  - » --memory="1000k"
  - » --memory="128m"
  - » --memory="1g"
- > --name="": 设置容器名称。
- > --net="bridge": 设置容器的网络模式。
  - » bridge: 在 Docker 网桥上创建新网络。
  - » none: 不使用网络。
  - » container:< 容器名, ID>: 一起使用其他容器的网络。
  - » host: 在容器内部使用主机网络。由于使用主机网络时可以通过 D-Bus 访问主机的所有系统访问, 所以人们认为容器是不安全的。
- > -P, --publish-all=false: 将连接到主机的容器的所有端口暴露在外。
- > -p, --publish=[]: 将连接到主机的容器的特定端口暴露在外。一般主要用于暴露 Web 服务器的端口。
  - » < 主机端口 >:< 容器端口 >: 如 -p 80:80。
  - » < IP 地址 >:< 主机端口 >:< 容器端口 >: 主机有多个网络接口或 IP 地址时使用。如 -p 192.168.0.10:80:80。
  - » < IP 地址 >:< 容器端口 >: 若未设置主机端口, 则随机设置主机端口号。如 -p 192.168.0.10::80。
  - » < 容器端口 >: 若只设置容器端口, 则随机设置主机端口号。如 -p 80。

- `--privileged=false`: 在容器内部使用主机的所有 Linux 内核功能 (Capability)。
- `--restart=""`: 设置容器内部进程终止时重启策略。
  - » `no`: 即使进程终止, 也不重启容器。如 `--restart="no"`。
  - » `on-failure`: 仅当进程的 Exit Code 不为 0 时执行重启。也可以设置重试次数。若不设置重试次数, 则不断重启。如 `--restart "on-failure:10"`。
  - » `always`: 不受进程的 Exit Code 影响, 总是重启。如 `--restart="always"`。
- `--security-opt=[]`: 设置 SELinux、AppArmor 选项。
  - » `--security-opt="label:level:TopSecret"`
- `-t`、`--tty=false`: 使用 TTY 模式 (pseudo-TTY)。若要使用 Bash, 则必须设置该选项。若不设置该选项, 则虽然可以输入命令, 但不显示 shell。
- `-u`、`--user=""`: 设置容器运行时要使用的 Linux 用户账户名与 UID。
- `-v`、`--volume=[]`: 设置数据卷。设置要与主机共享的目录, 不将文件保存到容器, 而直接保存到主机。在主机目录后添加 `:ro`、`:rw` 进行读写设置, 默认值为 `:rw`。更详细的内容请参考 6.4 节。
  - » < 容器目录 >: 如 `-v /data`。
  - » < 主机目录 >: < 容器目录 >: 如 `-v /data:/data`。
  - » < 主机目录 >: < 容器目录 >: < ro, rw >: 如 `-v /data:/data:ro`。
  - » < 主机文件 >: < 容器文件 >: 如 `-v /var/run/docker.sock:/var/run/docker.sock`。
- `--volumes-from=[]`: 连接数据卷容器, 设置格式为 < 容器名, ID >: < ro, rw >。默认情形下, 读写设置遵从 `-v` 选项的设置。更详细的内容请参考 6.5 节。
  - » `--volumes-from="hello"`
  - » 若设置为 `--volumes-from="hello:ro"`, 则数据卷为只读。
  - » 若设置为 `--volumes-from="hello:rw"`, 则数据卷为可读写。
- `-w`、`--workdir=""`: 设置容器内部要运行进程的目录。
  - » `--workdir="/var/www"`

运行如下命令, 创建容器。

---

```
$ sudo docker create -i -t --name hello ubuntu:14.04 /bin/bash
```

---

若想使用 `docker create` 命令创建的容器, 则必须使用 `docker start` 命令启动容器, 如下所示。

---

```
$ sudo docker start hello
$ sudo docker attach hello
root@dc6aeafab658:/#
```

---

## 19.6 ▶ diff

diff 命令用于检查容器文件系统的修改。

```
docker diff < 容器名称 , ID>
```

比较文件是否修改的标准是容器创建时的镜像内容。

- ▶ A: 添加的文件。
- ▶ C: 修改的文件。
- ▶ D: 删除的文件。

```
$ sudo docker run -i -t --name hello ubuntu:14.04 /bin/bash
root@22e5c01b5de1:/# echo "Hello World" > hello.txt
root@22e5c01b5de1:/# rm /usr/bin/yes
root@22e5c01b5de1:/# exit
$ sudo docker diff hello
A /.bash_history
A /hello.txt
C /usr
C /usr/bin
D /usr/bin/yes
```

由于创建了 hello.txt 文件，所以输出 A /hello.txt。同时，由于删除了 /usr/bin/yes 文件，所以输出 D /usr/bin/yes。删除 /usr/bin/yes 文件时，/usr/bin、/usr 目录也发生改动，故也显示 C /usr、C /usr/bin。并且，若在 shell 中输入命令，则 .bash\_history 文件也发生改变，所以也输出 A /.bash\_history。

## 19.7 ▶ events

events 命令用于实时输出 Docker 服务器中发生的事件。

```
docker events
```

- ▶ --since="": 输出特定 timestamp 之后的事件。
- ▶ --until="": 输出特定 timestamp 之前的事件。

运行 `docker events` 命令，进入待机状态。

---

```
$ sudo docker events
```

---

打开另一终端，运行容器，如下所示。

---

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
```

---

在执行 `docker events` 命令的终端中输出刚刚运行的命令的事件。

---

```
$ sudo docker events
```

```
2014-09-16T20:00:59+09:00 f873bbdfac50caccb1d6f78986ee33cced5d0e3869a8ef9c8c02c5bc9be5f766: (from
ubuntu:14.04) create
2014-09-16T20:00:59+09:00 f873bbdfac50caccb1d6f78986ee33cced5d0e3869a8ef9c8c02c5bc9be5f766: (from
ubuntu:14.04) start
```

---

在 `docker events` 命令中可以使用 `--since` 选项设置 Unix Timestamp 格式或日期。

---

```
$ sudo docker events --since '1410865200'
```

```
2014-09-16T20:00:59+09:00 f873bbdfac50caccb1d6f78986ee33cced5d0e3869a8ef9c8c02c5bc9be5f766: (from
ubuntu:14.04) create
2014-09-16T20:00:59+09:00 f873bbdfac50caccb1d6f78986ee33cced5d0e3869a8ef9c8c02c5bc9be5f766: (from
ubuntu:14.04) start
```

---



---

```
$ sudo docker events --since '2014-09-16'
```

```
2014-09-16T19:01:48+09:00 7bf16498ce57018d899c33f47f85ad550c682a2e44e7055e56addbbe974af1bf: (from
ubuntu:14.04) die
2014-09-16T19:01:48+09:00 7bf16498ce57018d899c33f47f85ad550c682a2e44e7055e56addbbe974af1bf: (from
ubuntu:14.04) destroy
```

---

## 19.8 > exec

`exec` 命令用于从外部运行容器内部的命令。

```
docker exec <选项> <容器名称, ID> <命令> <参数>
```

> `-d`、`--detach=false`: 以后台模式运行命令。

> `-i`、`--interactive=false`: 开启标准输入，即使未与容器连接，也维持标准输入。

➤ `-t`、`--tty=false`: 使用 TTY 模式 (pseudo-TTY)。若要使用 Bash, 则必须设置该选项。若不设置该选项, 则虽然可以输入命令, 但不显示 shell。

运行如下命令, 创建容器。

---

```
$ sudo docker run -d --name hello ubuntu:14.04 \
/bin/bash -c "while true; do echo Hello World; sleep 1; done"
```

---

设置每隔 1 秒输出 1 次 Hello World。此状态下, 运行容器内部的 `/bin/bash`, 连接至 Bash shell, 如下所示。连接 Bash shell 时, 只有使用 `-i -t` 选项才能输入命令并查看结果。

---

```
$ sudo docker exec -i -t hello /bin/bash
root@f31ddb5b0fa9:/# ps ax
 PID TTY STAT TIME COMMAND
 1 ? Ss 0:00 /bin/bash -c while true; do echo Hello World; sleep 1; done
 281 ? S 0:00 /bin/bash
 312 ? S 0:00 sleep 1
 313 ? R+ 0:00 ps ax
root@f31ddb5b0fa9:/# exit
```

---

若在容器内部运行 `ps ax` 命令, 则可以看到由 `docker exec` 命令运行的其他 `/bin/bash`, 与输出 Hello World 的 `/bin/bash` 不是同一个。输入 `exit` 命令退出 Bash shell 后, 容器不会停止, 而会继续运行。像这样, 灵活使用 `docker exec` 命令将 Bash shell 连接到正在运行守护进程的容器上, 并进行多种操作。

下面不连接 Bash shell, 而使用 `apt-get`、`yum` 等命令, 在容器内安装 `redis-server` 包, 如下所示。

---

```
$ sudo docker exec hello apt-get update
$ sudo docker exec hello apt-get install -y redis-server
```

---

如下所示, 也可以使用 `-d` 选项, 以后台方式运行命令 (进程)。此处以后台方式运行 `redis-server`。

---

```
$ sudo docker exec -d hello redis-server
$ sudo docker top hello ax
 PID TTY STAT TIME COMMAND
 10451 ? Ss 0:00 /bin/bash -c while true; do echo Hello World; sleep 1;
 10495 ? Ss 0:00 nsenter-exec --nspid 10451 -- redis-server
 10496 ? Sl 0:00 redis-server *:6379
 10745 ? S 0:00 sleep 1
```

---

## 19.9 ▶ export

export 命令用于将容器的文件系统导出为 tar 文件包。

```
docker export < 容器名称, ID>
```

只运行 docker export 命令后, 由于容器的内容会输出到标准输出, 所以必须设置重定向。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker export hello > hello.tar
```

## 19.10 ▶ history

history 命令用于显示镜像的历史。此处的历史依据 Dockerfile 文件中的设置创建。

```
docker history < 选项 >< 镜像名称, ID>
```

> --no-trunc=false: 输出所有因内容过长而省略的部分。

> -q、--quiet=false: 只显示镜像 ID。

```
$ echo -e "FROM ubuntu:14.04\nRUN apt-get update" | sudo docker build -t hello -
```

```
$ sudo docker history hello
```

| IMAGE        | CREATED        | CREATED BY                                        | SIZE     |
|--------------|----------------|---------------------------------------------------|----------|
| 53b7ff32bee2 | 19 seconds ago | /bin/sh -c apt-get update                         | 20.29 MB |
| 826544226fdc | 11 days ago    | /bin/sh -c #(nop) CMD [/bin/bash]                 | 0 B      |
| c7c7108e0ad8 | 11 days ago    | /bin/sh -c apt-get update && apt-get dist-upg     | 1.472 MB |
| 7428bd008763 | 11 days ago    | /bin/sh -c sed -i 's/^#\s*\((deb.*universe\) \$/' | 1.895 kB |
| ff01d67c9471 | 11 days ago    | /bin/sh -c rm -rf /var/lib/apt/lists/*            | 0 B      |
| ca63a3899a99 | 11 days ago    | /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic     | 194.5 kB |
| b3553b91f79f | 11 days ago    | /bin/sh -c #(nop) ADD file:fafe77ac6fd8eb555b     | 192.5 MB |
| 511136ea3c5a | 15 months ago  |                                                   | 0 B      |

由于 hello 镜像的 Dockerfile 文件中运行了 apt-get update 命令, 所以历史中也显示 /bin/sh -c apt-get update。



## 19.11 ▶ images

image 命令用于输出镜像列表。

```
docker images <选项><镜像名称>
```

- -a、--all=false: 列出所有镜像，包括父镜像。
- -f、--filter=[]: 设置输出结果过滤。若设置为 "dangling=true", 则只输出无名镜像。
- --no-trunc=false: 显示所有因内容过长而省略的部分。
- -q、--quiet=false: 只输出镜像 ID。

若在 docker images 命令中设置镜像名称，则只输出名称相同而标签不同的镜像。

```
$ sudo docker images ubuntu
```

| REPOSITORY | TAG   | IMAGE ID     | CREATED     | VIRTUAL SIZE |
|------------|-------|--------------|-------------|--------------|
| ubuntu     | 14.04 | 826544226fdc | 11 days ago | 194.2 MB     |
| ubuntu     | 12.04 | c17f3f519388 | 11 days ago | 106.7 MB     |

如下所示，在 -f 选项中设置 "dangling=true", 只输出无名镜像。

```
$ sudo echo -e "FROM ubuntu:14.04\nRUN apt-get update" | sudo docker build -
$ sudo docker images -f "dangling=true"
```

| REPOSITORY | TAG    | IMAGE ID     | CREATED        | VIRTUAL SIZE |
|------------|--------|--------------|----------------|--------------|
| <none>     | <none> | 47477f7f6e79 | 13 seconds ago | 214.5 MB     |

若要删除所有无名镜像，运行如下命令即可。

```
$ sudo docker rmi $(sudo docker images -f "dangling=true" -q)
```

## 19.12 ▶ import

import 命令用于从压缩为 tar 文件 (.tar、.tar.gz、.tgz、.bzip、.tar.xz、.txz) 的文件系统创建镜像。

```
docker import <tar 文件的 URL 或者 -><注册名称>/<镜像名称>:<标签>
```

使用 `import` 命令时，可以设置 `tar` 文件的 URL，若设置为 `-`，则从标准输入接收 `tar` 文件的内容。既可以使用由 `docker export` 命令创建的 `tar` 文件，也可以直接组织文件系统。关于组织文件系统并创建基础镜像的方法请参考 6.6 节。

下列命令使用网络上的 `tar` 文件的 URL 创建镜像。

---

```
$ sudo docker import http://example.com/hello.tar.gz hello
```

---

下列命令中，将位于本地的 `hello.tar` 文件的内容通过管道传递给 `docker import` 命令，以创建镜像。

---

```
$ cat hello.tar | docker import - hello
```

---

若想将当前目录的内容直接创建为镜像，则要运行如下命令。

---

```
$ sudo tar -c . | sudo docker import - hello
```

---

## 19.13 ▶ info

`info` 命令用于显示当前系统信息、Docker 容器、镜像个数、设置等信息。

`docker info`

---

```
$ sudo docker info
```

```
Containers: 3
```

```
Images: 13
```

```
Storage Driver: aufs
```

```
Root Dir: /var/lib/docker/aufs
```

```
Dirs: 19
```

```
Execution Driver: native-0.2
```

```
Kernel Version: 3.13.0-24-generic
```

```
Operating System: Ubuntu 14.04 LTS
```

```
Username: exampleuser
```

```
Registry: [https://index.docker.io/v1/]
```

```
WARNING: No swap limit support
```

---

## 19.14 inspect

inspect 命令用于以 JSON 格式显示容器与镜像的详细信息。

```
docker inspect < 选项 >< 容器或镜像名称, ID>
```

➤ -f、--format="": 只显示指定信息。像 "{.NetworkSettings.IPAddress}" 一样, 可以使用 . (点) 设置 JSON 文档的下层项目。

以下命令只用于在镜像的详细信息中显示架构与 OS 相关信息。

```
$ sudo docker inspect -f "{{.Architecture}} {{.Os }}" ubuntu:14.04
amd64 linux
```

下列命令用于显示容器的 IP 地址。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker inspect -f "{{.NetworkSettings.IPAddress }}" hello
172.17.0.85
```

以下命令采用 JSON 格式显示一部分详细信息。

```
$ sudo docker inspect -f "{{json .NetworkSettings }}" hello
{"Bridge":"docker0","Gateway":"172.17.42.1","IPAddress":"172.17.0.85","IPPrefixLen":16,"PortMapping":
null,"Ports":{"80/tcp":[{"HostIp":"0.0.0.0","HostPort":"80"}],"8080/tcp"}
```

下列命令只从容器的详细信息中抽取特定部分, 并按照所希望的格式显示。

```
$ sudo docker run -i -t -d --name hello -p 80:80 -p 8080:8080 ubuntu:14.04 /bin/bash
$ sudo docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{ $p}} -> {{(index $conf
0).HostPort}} {{end}}' hello
80/tcp -> 80 8080/tcp -> 8080
```

.NetworkSettings.Ports 的内容如下所示。

```
"Ports": {
 "80/tcp": [
 {
 "HostIp": "0.0.0.0",
 "HostPort": "80"
 }
]
}
```

```

],
"8080/tcp": [
 {
 "HostIp": "0.0.0.0",
 "HostPort": "8080"
 }
]
}

```

此处使用 `{{range $p, $conf := .NetworkSettings.Ports}}` 循环访问 `.NetworkSettings.Ports` 的内容，并代入 `$p`、`$conf`。然后如实输出 `$p`，并将 `$conf` 数组的第一项 (`index $conf 0`) 的 `.HostPort` 输出。

```

{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}

```

## 19.15 > kill

`kill` 命令用于向容器发送 KILL 信号，从而关闭容器。

```
docker kill <选项> <容器名称, ID>
```

> `-s`、`--signal="KILL"`：发送特定信号。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
```

```
$ sudo docker ps
```

| CONTAINER ID | IMAGE        | COMMAND     | CREATED       | STATUS       | PORTS | NAMES |
|--------------|--------------|-------------|---------------|--------------|-------|-------|
| 0660ca3469bf | ubuntu:14.04 | "/bin/bash" | 3 seconds ago | Up 2 seconds |       | hello |

```
$ sudo docker kill hello
```

```
$ sudo docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------|---------|---------|--------|-------|-------|
|--------------|-------|---------|---------|--------|-------|-------|

## 19.16 > load

`load` 命令用于从 tar 文件创建镜像。

```
docker load <选项>
```

将 tar 文件发送到 `docker load` 命令的标准输入，然后创建镜像。tar 文件由 `docker save` 命令创建，包含镜像名称与标签。

➤ `-i, --input=""`: 不使用标准输入，设置文件路径并创建镜像。

下面先使用 `docker save` 命令将 `ubuntu:12.04`、`ubuntu:14.04` 镜像保存为 `ubuntu.tar` 文件，然后使用 `docker load` 命令创建为镜像。

---

```
$ sudo docker images
```

| REPOSITORY | TAG   | MAGE ID      | CREATED     | VIRTUAL SIZE |
|------------|-------|--------------|-------------|--------------|
| ubuntu     | 14.04 | 826544226fdc | 11 days ago | 194.2 MB     |
| ubuntu     | 12.04 | c17f3f519388 | 11 days ago | 106.7 MB     |

```
$ sudo docker save ubuntu > ubuntu.tar
$ sudo docker rmi `docker images -aq`
$ sudo docker images
```

| REPOSITORY | TAG   | MAGE ID      | CREATED     | VIRTUAL SIZE |
|------------|-------|--------------|-------------|--------------|
| ubuntu     | 14.04 | 826544226fdc | 11 days ago | 194.2 MB     |
| ubuntu     | 12.04 | c17f3f519388 | 11 days ago | 106.7 MB     |

```
$ sudo docker load < ubuntu.tar
$ sudo docker images
```

| REPOSITORY | TAG   | MAGE ID      | CREATED     | VIRTUAL SIZE |
|------------|-------|--------------|-------------|--------------|
| ubuntu     | 14.04 | 826544226fdc | 11 days ago | 194.2 MB     |
| ubuntu     | 12.04 | c17f3f519388 | 11 days ago | 106.7 MB     |

---

## 19.17 ▶ login

`login` 命令用于登录 Docker 的注册服务器。

```
docker login < 选项 ><Docker 注册服务器的 URL>
```

若不设置注册服务器的地址，则默认登录 <https://index.docker.io/v1/>。

- `-e, --email=""`: 设置登录时使用的电子邮件。
- `-p, --password=""`: 设置登录时使用的密码。
- `-u, --username=""`: 设置登录时使用的 Docker 注册服务器账号。

---

```
$ sudo docker login
Username: exampleuser
Password: <输入密码>
Email: exampleuser@example.com
Login Succeeded
```

---

## 19.18 > logout

logout 命令用于从 Docker 注册服务器中登出。

```
docker logout <Docker 注册服务器 URL>
```

若不设置注册服务器地址，则默认从 <https://index.docker.io/v1/> 中登出。

---

```
$ sudo docker logout
```

```
Remove login credentials for https://index.docker.io/v1/
```

---

## 19.19 > logs

logs 命令用于输出容器日志。

```
docker logs <容器名称, ID>
```

虽然使用 `docker attach` 命令也可以输入，但使用 `docker logs` 命令只能输出日志。

- > `-f`、`--follow=false`：一直输出实时日志。
- > `-t`、`--timestamp=false`：在登录前显示时间值。
- > `--tail="all"`：指定数字，只从最终日志中输出一定个数。

---

```
$ sudo docker run -d --name hello ubuntu:14.04 \
 /bin/bash -c "while true; do echo Hello World; sleep 1; done"
```

```
$ sudo docker logs hello
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

---



## 19.20 ▶ port

port 命令用于查看容器的某个端口是否处于开放状态。

```
docker port < 容器名称 , ID> < 端口 >
```

```
$ sudo docker run -i -t -d --name hello -p 80:80 ubuntu:14.04 /bin/bash
$ sudo docker port hello 80
0.0.0.0:80
```

## 19.21 ▶ pause

pause 命令用于暂停容器中正在运行的所有进程。

```
docker pause < 容器名称 , ID>
```

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker pause hello
```

| CONTAINER ID | IMAGE        | COMMAND     | CREATED        | STATUS                 | PORTS | NAMES |
|--------------|--------------|-------------|----------------|------------------------|-------|-------|
| 546073b75f7c | ubuntu:14.04 | "/bin/bash" | 12 seconds ago | Up 12 seconds (Paused) |       | hello |

## 19.22 ▶ ps

ps 命令用于输出容器列表。

```
docker ps < 选项 >
```

- ▶ -a、--all=false: 列出所有容器。使用 docker ps 命令只列出默认启动的容器。
- ▶ --before="": 列出特定容器创建前创建的容器，包含停止的容器。
- ▶ -f、--filter=: 设置输出过滤。如 "exited=0"。
- ▶ -l、--latest=false: 列出最后创建的容器，包含停止的容器。

- `-n=-1`: 设置数字, 只从最近创建的容器中显示一定个数, 包含停止的容器。
- `--no-trunc=false`: 列出所有因内容过长而省略的部分。
- `-q`、`--quiet=false`: 只列出容器 ID。
- `-s`、`--size=false`: 显示容器中变更数据的大小。
- `--since=""`: 显示特定容器创建后创建的容器, 包含停止的容器。

下面在 `docker ps` 命令中使用 `-a` 选项, 列出所有容器, 包含停止的容器。

```
$ sudo docker run -i -t -d --name hello1 ubuntu:14.04 /bin/bash
$ sudo docker run -i -t -d --name hello2 ubuntu:14.04 /bin/bash
$ sudo docker stop hello1
$ sudo docker ps -a
```

| CONTAINER ID | IMAGE        | COMMAND     | CREATED        | STATUS                    | PORTS | NAMES  |
|--------------|--------------|-------------|----------------|---------------------------|-------|--------|
| 23e001ed649b | ubuntu:14.04 | "/bin/bash" | 22 seconds ago | Up 21 seconds             |       | hello2 |
| eca06f6e32b0 | ubuntu:14.04 | "/bin/bash" | 30 seconds ago | Exited (0) 14 seconds ago |       | hello1 |

## 19.23 ➤ pull

`pull` 命令用于从 Docker 注册服务器下载镜像。

```
docker pull <选项> <注册名称> / <镜像名称> : <标签>
```

- `-a`、`--all-tags=false`: 下载镜像的所有标签。
- 注册名称中既可以设置 Docker Hub 用户名, 也可以设置注册地址。
- 官方镜像省略 Docker Hub 用户名, 只设置镜像名。
- 若不设置标签, 则下载所有带标签的镜像。

```
$ sudo docker pull centos
$ sudo docker pull ubuntu:14.04
$ sudo docker pull registry.hub.docker.com/ubuntu:14.04
$ sudo docker pull exampleuser/hello:0.1
```

运行如下命令, 从个人仓库下载镜像。

```
$ sudo docker pull 192.168.0.39:5000/hello:0.1
$ sudo docker pull exampleregistry.com:5000/hello:0.1
```

## 19.24 ▶ push

push 命令用于将镜像推送到 Docker 注册服务器。

```
docker push <注册名>/<镜像名>:<标签>
```

- ▶ 注册名称中既可以设置 Docker Hub 用户名，也可以设置注册地址。
- ▶ 若不设置标签，则推送所有带标签的镜像。

```
$ sudo docker tag hello:0.1 exampleuser/hello:0.1
$ sudo docker pull exampleuser/hello:0.1
```

使用如下命令将镜像推送到个人仓库。

```
$ sudo docker tag hello:0.1 192.168.0.39:5000/hello:0.1
$ sudo docker pull 192.168.0.39:5000/hello:0.1
$ sudo docker tag hello:0.1 exampleregistry.com:5000/hello:0.1
$ sudo docker pull exampleregistry.com:5000/hello:0.1
```

## 19.25 ▶ restart

restart 命令用于重启容器。

```
docker restart <选项><容器名称, ID>
```

- ▶ -t、--time=10: 设置从容器停止到重启的等待时间，单位是秒。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker restart hello
```

## 19.26 ▶ rm

rm 命令用于删除容器。

```
docker rm < 选项 >< 容器名称, ID>
```

- > -f、--force=false: 强制停止容器后删除（使用 SIGKILL 信号）。
- > -l、--link=false: 在 docker run 命令中使用 --link 选项，只删除连接，不删除容器。
- > -v、--volumes=false: 删除连接到容器的数据卷。

下列命令只用于删除容器间的连接。

```
$ sudo docker run -i -t -d --name db ubuntu:14.04 /bin/bash
$ sudo docker run -i -t -d --name hello --link db:db ubuntu:14.04 /bin/bash
$ sudo docker ps
```

| CONTAINER ID | IMAGE        | COMMAND     | CREATED       | STATUS       | PORTS | NAMES       |
|--------------|--------------|-------------|---------------|--------------|-------|-------------|
| 71f8570c3683 | ubuntu:14.04 | "/bin/bash" | 1 seconds ago | Up 1 seconds |       | hello       |
| 61f65b7cccd  | ubuntu:14.04 | "/bin/bash" | 5 seconds ago | Up 4 seconds |       | db,hello/db |

```
$ sudo docker rm -l hello/db
$ sudo docker ps
```

| CONTAINER ID | IMAGE        | COMMAND     | CREATED        | STATUS        | PORTS | NAMES |
|--------------|--------------|-------------|----------------|---------------|-------|-------|
| 71f8570c3683 | ubuntu:14.04 | "/bin/bash" | 19 seconds ago | Up 18 seconds |       | hello |
| 61f65b7cccd  | ubuntu:14.04 | "/bin/bash" | 23 seconds ago | Up 22 seconds |       | db    |

若要一次删除所有容器，如下所示，在 docker ps 命令中使用 -a、-q 选项获取容器 ID，之后传递给 docker rm 命令。

```
$ sudo docker rm $(sudo docker ps -aq)
$ sudo docker rm $(sudo docker ps -aq)
```

类似 -a、-q 的短选项可以彼此结合使用。

## 19.27 > rmi

rmi 命令用于删除镜像。若不指定标签，则删除 latest 标签。

```
docker rmi < 注册名称 >/< 镜像名称, ID>[:< 标签 >]
```

- > -f、--force=false: 强制删除镜像。
- > --no-prune=false: 不删除不带标签的父镜像。

```
$ sudo docker rm hello
$ sudo docker rm hello:0.1
$ sudo docker rm exampleuser/hello:0.1
$ sudo docker rm 192.168.0.39:5000/hello:0.1
$ sudo docker rm exampleregistry.com:5000/hello:0.1
```

使用如下命令强制删除正在运行的镜像。

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
$ sudo docker rm -f hello
```

若想一次删除所有镜像，如下所示，在 `docker images` 命令中使用 `-a`、`-q` 选项获取镜像 ID，然后传递给 `docker rmi` 命令。

```
$ sudo docker rmi $(sudo docker images -aq)
$ sudo docker rmi $(sudo docker images -aq)
```

类似 `-a`、`-q` 的短选项可以彼此结合使用。

## 19.28 ▶ run

`run` 命令用于使用指定镜像创建容器。

```
docker run <选项> <镜像名称, ID> <命令> <参数>
```

设置选项值时，可以省略 `=` 与 `"`。

- ▶ `-a`、`--attach[]`：将标准输入、标准输出、标准错误连接到容器。
  - » `--attach="stdin"`
- ▶ `--add-host=[]`：向容器的 `/etc/hosts` 添加主机名与 IP 地址。
  - » `--add-host=hello:192.168.0.10`
- ▶ `-c`、`--cpu-shares=0`：设置 CPU 资源分配，默认设置值为 1024，各设置值为相对值。
  - » 若设置为 `--cpu-shares=2048`，则分配默认值 2 倍的 CPU 资源。
  - » 在 Linux 内核的 `cgroups` 中使用该设置值。
- ▶ `--cap-add[]`：设置容器中使用 `cgroups` 的特定 Capability。若设置为 `ALL`，则使用所有 Capability。
  - » 像 `--cap-add="MKNOD"` `--cap-add="NET_ADMIN"` 一样进行设置。关于所有 Capability 列表请

参考如下链接。

<http://linux.die.net/man/7/capabilities>

- `--cap-drop=[]` : 从容器删除 cgroups 的特定 Capability。
- `--cidfile=""` : 设置 cid 文件路径。cid 中存储着所创建容器的 ID。
- `--cpuset=""` : 在多核 CPU 中设置要运行容器的核心。
  - » 若设置为 `--cpuset="0, 1"`, 则使用第一个与第二个 CPU 内核。
  - » 若设置为 `--cpuset="0-3"`, 则使用从第一个到第三个 CPU 内核。
- `d, --detach=false` : Detached 模式, 一般称为守护进程模式, 容器以后台方式运行。
- `--device=[]` : 添加主机设备到容器, 格式为 `< 主机设备 >: < 容器设备 >`。
  - » 若设置为 `--device="/dev/sda1:/dev/sda1"`, 则在容器也可以使用主机的 `/dev/sda1` 块设备。
- `--dns=[]` : 设置容器中要使用的 DNS 服务器。
  - » `--dns="8.8.8.8"`
- `--dns-search=[]` : 设置容器中要使用的 DNS 搜索域。
  - » 若设置为 `--dns-search="example.com"`, 则向 DNS 查询 hello 时, 优先查找 hello.example.com。
- `-e, --env=[]` : 向容器设置环境变量。一般用于传递设置值或密码。
  - » `-e MYSQL_ROOT_PASSWORD=examplepassword`
- `--entrypoint=""` : 忽略 Dockerfile 的 ENTRYPOINT 设置, 强制设置为其他值。
  - » `--entrypoint="/bin/bash"`
- `--env-file=[]` : 向容器应用设有环境变量的文件。
  - » `--env-file="/etc/environment"`
- `--expose=[]` : 仅连接容器的端口与主机, 并不暴露在外。
  - » `--expose="3306"`
- `-h, --hostname=""` : 设置容器的主机名。
- `-i, --interactive=false` : 激活标准输入, 即使未与容器连接, 也维持标准输入。一般使用该选项在 Bash 中输入命令。
- `--link=[]` : 进行容器间连接, 格式为 `< 容器名称 >: < 别名 >`。
  - » `--link="db:db"`
- `--lxc-conf=[]` : 若使用 LXC 驱动, 则可以设置 LXC 选项。
  - » `--lxc-conf="lxc.cgroup.cpuset.cpus = 0,1"`
- `-m, --memory=""` : 设置内存限制, 格式为 `< 数字 > < 单位 >`, 单位可以使用 b、k、m、g。
  - » `--memory="1000000b"`
  - » `--memory="1000k"`
  - » `--memory="128m"`



- » `--memory="1g"`
- » `--name=""`: 设置容器名。
- » `--net="bridge"`: 设置容器的网络模式。
  - » `bridge`: 在 Docker 网桥上创建新网络。
  - » `none`: 不使用网络。
  - » `container:< 容器名, ID>`: 一起使用其他容器的网络。
  - » `host`: 在容器内部使用主机网络。使用主机网络时可以通过 D-Bus 访问主机的所有系统访问, 所以人们认为容器是不安全的。
- » `-P, --publish-all=false`: 将连接到主机的容器的所有端口暴露在外。
- » `-p, --publish=[]`: 将连接到主机的容器的特定端口暴露在外。一般主要用于暴露 Web 服务器的端口。
  - » `< 主机端口 >:< 容器端口 >`: 如 `-p 80:80`。
  - » `< IP 地址 >:< 主机端口 >:< 容器端口 >`: 有多个主机网络接口或 IP 地址时使用。如 `-p 192.168.0.10:80:80`。
  - » `< IP 地址 >::< 容器端口 >`: 若未设置主机端口, 则随机设置主机端口号。如 `-p 192.168.0.10::80`。
  - » `< 容器端口 >`: 若只设置容器端口, 则随机设置主机端口号。如 `-p 80`。
- » `--privileged=false`: 在容器内部使用主机的所有 Linux 内核功能 (Capability)。
- » `--restart=""`: 设置容器内部进程终止时重启策略。
  - » `no`: 即使进程终止, 也不重启容器。如 `--restart="no"`。
  - » `on-failure`: 仅当进程的 Exit Code 不为 0 时执行重启。也可以设置重试次数。若不设置重试次数, 则不断重启。如 `--restart "on-failure:10"`。
  - » `always`: 不受进程的 Exit Code 影响, 总是重启。如 `--restart="always"`。
- » `--rm=false`: 若容器内的进程终止, 则自动删除容器。该选项不能与 `-d` 选项一起使用。
- » `--security-opt=[]`: 设置 SELinux、AppArmor 选项。
  - » `--security-opt="label:level:TopSecret"`
- » `--sig-proxy=true`: 将所有信号传递给进程 (非 TTY 模式时也一样), 但不传递 SIGCHLD、SIGKILL、SIGSTOP 信号。
- » `-t, --tty=false`: 使用 TTY 模式 (pseudo-TTY)。若要使用 Bash, 则必须设置该选项。若不设置该选项, 则虽然可以输入命令, 但不显示 shell。
- » `-u, --user=""`: 设置容器运行时要使用的 Linux 用户账户名与 UID。
- » `-v, --volume=[]`: 设置数据卷。设置要与主机共享的目录, 不将文件保存到容器, 而直接保存到主机。在主机目录后添加 `:ro`、`:rw` 进行读写设置, 默认值为 `:rw`。更详细的内容请参考 6.4 节。

» < 容器目录 >: 如 -v /data。

» < 主机目录 >:< 容器目录 >: 如 -v /data:/data。

» < 主机目录 >:< 容器目录 >:<ro, rw>: 如 -v /data:/data:ro。

» < 主机文件 >:< 容器文件 >: 如 -v /var/run/docker.sock:/var/run/docker.sock。

» --volumes-from=[]: 连接数据卷容器, 设置格式为 < 容器名, ID>:<ro, rw>。默认情形下, 读写设置遵从 -v 选项的设置。更详细的内容请参考 6.5 节。

» --volumes-from="hello"

» 若设置为 --volumes-from="hello:ro", 则数据卷为只读。

» 若设置为 --volumes-from="hello:rw", 则数据卷为可读写。

» -w、--workdir="": 设置容器内部要运行进程的目录。

» --workdir="/var/www"

**提示** =[] 用于表示设置选项时可以设置为不同值, 并且可以多次使用。以 -p、--publish=[] 为例, 用法如下所示。

```
$ sudo docker run -d -p 80:80 -p 443:443 nginx:latest
```

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@2608a5be6bc0:/#
```

下面使用 -i、-a 选项将 "Hello World" 字符串发送到容器内 cat 命令的标准输入。若使用 docker logs 命令输出日志, 将显示 cat 命令输出的字符串。

```
$ ID=$(echo "Hello World" | sudo docker run -i -a stdin ubuntu:14.04 cat -)
$ sudo docker logs $ID
Hello World
```

下列设置中, 使用 --cap-add 选项使得在容器内部可以使用 SYS\_ADMIN Capability。

```
$ sudo docker run -it --rm ubuntu:14.04 bash
root@1bcbd8bc059b:/# mount -t tmpfs none /mnt
mount: permission denied
root@1bcbd8bc059b:/# exit
$ sudo docker run -it --rm --name hello --cap-add SYS_ADMIN ubuntu:14.04 bash
root@c58569c2175c:/# mount -t tmpfs none /mnt
root@c58569c2175c:/# df -h
```

| Filesystem | Size | Used | Avail | Use% | Mounted on |
|------------|------|------|-------|------|------------|
| rootfs     | 28G  | 2.0G | 24G   | 8%   | /          |
| none       | 28G  | 2.0G | 24G   | 8%   | /          |
| tmpfs      | 2.0G | 0    | 2.0G  | 0%   | /dev       |

|                             |      |      |      |    |             |
|-----------------------------|------|------|------|----|-------------|
| shm                         | 64M  | 0    | 64M  | 0% | /dev/shm    |
| /dev/mapper/ubuntu--vg-root | 28G  | 2.0G | 24G  | 8% | /etc/hosts  |
| tmpfs                       | 2.0G | 0    | 2.0G | 0% | /proc/kcore |
| none                        | 2.0G | 0    | 2.0G | 0% | /mnt        |

下列命令中，使用 `-p` 选项将容器的 8080 端口连接到主机 192.168.0.10 的 80 端口，并暴露在外。

```
$ sudo docker -p 192.168.0.10:80:8080 ubuntu:14.04 bash
```

下面命令使用 `--expose` 选项只将 80 端口连接至主机，并未将其暴露在外。这样就无法从外部进行连接，而只能从与主机连接（使用 `--link` 选项）的容器中访问。

```
$ sudo docker --expose 80 ubuntu:14.04 bash
```

下面使用 `-e` 选项将环境变量 `HELLO_VAR` 设置到容器。

```
$ sudo docker -it -e HELLO_VAR="Hello World" ubuntu:14.04 bash
root@01ab56801da3:/# echo $HELLO_VAR
Hello World
```

编写环境变量文件，如下所示。

### > example-env.sh

```
HELLO=1234
WORLD=abcd
EXAMPLE
```

若使用 `--env-file` 选项设置环境变量文件，则可以使用容器内设置的环境变量。使用 `--env-file` 选项获取的环境变量名与使用 `-e` 选项设置的环境变量名相同时，`-e` 选项会覆盖之前的环境变量值。

```
$ sudo docker run -it \
 --env-file ./example-env.sh -e HELLO="Hello World" ubuntu:14.04 bash
root@2778675cf83e:/# echo $HELLO
Hello World
root@2778675cf83e:/# echo $WORLD
abcd
```

如下所示，未在 `example-env.sh` 文件中设置类似 `EXAMPLE` 的值时，可以不使用 `-e` 选项，而使用进程的环境变量设置值。

```
$ sudo EXAMPLE=10 docker run -it --env-file ./example-env.sh ubuntu:14.04 bash
root@de90b5dd1419:/# echo $EXAMPLE
10
```

在下列命令中使用 `--link` 选项连接 Redis 容器。在其他容器中使用别名 `cache` 可以访问 Redis 容器。

```
$ sudo docker run -d --name cache redis:latest
$ sudo docker run -it --link cache:cache ubuntu:14.04 bash
root@958359f0569f:/# ping cache
PING cache (172.17.0.157) 56(84) bytes of data.
64 bytes from cache (172.17.0.157): icmp_seq=1 ttl=64 time=0.047 ms
64 bytes from cache (172.17.0.157): icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from cache (172.17.0.157): icmp_seq=3 ttl=64 time=0.062 ms
```

## 19.29 > save

`save` 命令用于将镜像保存为 tar 包文件。

```
docker save < 选项 > < 镜像名称 >:< 标签 >
```

> `-o`、`--output=""`：设置要保存的文件名。

若不设置 `-o` 选项，tar 文件会输出到标准输出，所以必须设置重定向。如果仅指定镜像名称而未指定标签，则将所有标签保存到一个 tar 文件。

```
$ sudo docker save -o nginx.tar nginx:latest
$ sudo docker save -o redis.tar redis:latest
$ sudo docker save ubuntu:14.04 > ubuntu14.04.tar
$ sudo docker save ubuntu > ubuntu.tar
```

使用 `docker load` 命令可以再次从 tar 包加载镜像。

```
$ sudo docker load < ubuntu.tar
$ sudo docker images
```

| REPOSITORY | TAG   | IMAGE ID     | CREATED     | VIRTUAL SIZE |
|------------|-------|--------------|-------------|--------------|
| ubuntu     | 14.04 | 826544226fdc | 11 days ago | 194.2 MB     |
| ubuntu     | 12.04 | c17f3f519388 | 11 days ago | 106.7 MB     |

## 19.30 ▶ search

search 命令用于在 Docker Hub 中搜索镜像。

```
docker search < 选项 >< 搜索词 >
```

- ▶ `--automated=false`: 只显示由 Docker Hub 的 Automated Build 创建的镜像。
- ▶ `--no-trunc=false`: 显示所有因内容过长而省略的部分。
- ▶ `-s`、`--stars=0`: 显示带有特定星级以上的镜像。

```
$ sudo docker search -s 10 ubuntu
```

| NAME                               | DESCRIPTION                                   | STARS | OFFICIAL | AUTOMATED |
|------------------------------------|-----------------------------------------------|-------|----------|-----------|
| ubuntu                             | Official Ubuntu base image                    | 658   | [OK]     |           |
| dockerfile/ubuntu                  | Trusted automated Ubuntu (http://www.ubunt... | 23    |          | [OK]      |
| crashsystems/gitlab-docker         | A trusted, regularly updated build of GitL... | 20    |          | [OK]      |
| ubuntu-upstart                     | Upstart is an event-based replacement for ... | 13    | [OK]     |           |
| mbentley/ubuntu-django-uwsgi-nginx |                                               | 12    |          | [OK]      |
| dockerfile/ubuntu-desktop          | Trusted automated Ubuntu Desktop (LXDE) (h... | 11    |          | [OK]      |

## 19.31 ▶ start

start 命令用于启动容器。

```
docker start < 选项 >< 容器名称, ID>
```

- ▶ `-a`、`--attach=false`: 将标准输入、标准输出、标准错误连接到容器，传递所有信号。
- ▶ `-i`、`--interactive=false`: 激活标准输入。

下列命令用于强制终止由守护进程模式创建的容器，使用 `-a`、`-i` 选项显示输出内容。

```
$ sudo docker run -d --name hello ubuntu:14.04 \
 /bin/bash -c "while true; do echo Hello World; sleep 1; done"
$ sudo docker kill hello
$ sudo docker start -a -i hello
Hello World
Hello World
Hello World
```

## 19.32 ▶ stop

stop 命令用于终止容器。

```
docker stop < 选项 >< 容器名称, ID>
```

▶ -t、--time=10: 设置终止容器前的等待时间, 单位为秒。

下列命令中, 将 -t 选项设置为 0, 不等待, 直接终止运行中的容器。

```
$ sudo docker run -d --name hello ubuntu:14.04 \
 /bin/bash -c "while true; do echo Hello World; sleep 1; done"
$ sudo docker stop -t 0 hello
```

## 19.33 ▶ tag

tag 命令用于为镜像设置标签。

```
docker tag < 选项 >< 镜像名称 >:< 标签 >< 注册地址, 用户名 >/< 镜像名称 >:< 标签 >
```

▶ -f、--force=false: 即使已经拥有标签也强制设置。

使用 docker push 命令将镜像推送到 Docker Hub 或个人仓库时, 必须如下设置标签。

```
$ echo "FROM ubuntu:14.04" | sudo docker build -t hello:latest -
$ sudo docker tag hello:latest hello:0.1
$ sudo docker tag hello:latest exampleuser/hello:0.1
$ sudo docker tag hello:latest 192.168.0.39/hello:0.1
$ sudo docker images
```

| REPOSITORY         | TAG    | IMAGE ID     | CREATED     | VIRTUAL SIZE |
|--------------------|--------|--------------|-------------|--------------|
| exampleuser/hello  | 0.1    | 826544226fdc | 12 days ago | 194.2 MB     |
| 192.168.0.39/hello | 0.1    | 826544226fdc | 12 days ago | 194.2 MB     |
| hello              | 0.1    | 826544226fdc | 12 days ago | 194.2 MB     |
| hello              | latest | 826544226fdc | 12 days ago | 194.2 MB     |
| ubuntu             | 14.04  | 826544226fdc | 12 days ago | 194.2 MB     |
| ubuntu             | 12.04  | c17f3f519388 | 12 days ago | 106.7 MB     |



## 19.34 ▶ top

top 命令用于显示容器中正在运行的进程信息。

```
docker top <容器名称, ID><ps 选项>
```

在 <ps 选项> 中设置 Linux ps 命令的选项。

```
$ sudo docker run -d --name hello redis:latest
```

```
$ sudo docker top hello aux
```

| USER | PID   | %CPU | %MEM | VSZ   | RSS  | TTY | STAT | START | TIME | COMMAND             |
|------|-------|------|------|-------|------|-----|------|-------|------|---------------------|
| 999  | 14363 | 0.2  | 0.1  | 36432 | 7588 | ?   | Ssl  | 14:49 | 0:00 | redis-server *:6379 |

### 提示 ps 选项

- -a: 显示所有用户进程。
- -u: 显示各进程的用户 (UID)。
- -s: 显示信号。
- -v: 显示虚拟内存。
- -x: 显示无控制终端的进程。
- -c: 显示内核中使用的进程名称。
- -e: 显示环境变量。
- -f: 以完整形式显示内容。
- -l: 以详细格式显示。
- -n: 以数字形式显示 WCHAN 值。

## 19.35 ▶ unpause

unpause 命令用于重启 pause 命令暂停的容器。

```
docker unpause <容器名称, ID>
```

```
$ sudo docker run -i -t -d --name hello ubuntu:14.04 /bin/bash
```

```
$ sudo docker pause hello
```

```
$ sudo docker unpause hello
```

## 19.36 ▶ version

version 命令用于输出 Docker 的版本信息。

```
docker version
```

```
$ sudo docker version
```

```
Client version: 1.2.0
```

```
Client API version: 1.14
```

```
Go version (client): go1.3.1
```

```
Git commit (client): fa7b24f
```

```
OS/Arch (client): linux/amd64
```

```
Server version: 1.2.0
```

```
Server API version: 1.14
```

```
Go version (server): go1.3.1
```

```
Git commit (server): fa7b24f
```

## 19.37 ▶ wait

wait 命令等待容器终止，然后输出 Exit Code。

```
docker wait < 容器名称, ID>
```

运行如下命令，创建 Redis 容器。如果运行 docker wait 命令，则进入待机状态。

```
$ sudo docker run -d --name hello redis:latest
```

```
$ sudo docker wait hello
```

运行另一终端，强制终止 hello 容器。

```
$ sudo docker kill hello
```

运行 docker wait 命令的终端显示 Exit Code。

```
$ sudo docker wait hello
```

```
-1
```

# 编译 Docker

与其他项目不同，Docker 的编译也要在 Docker 容器中进行。编译 Docker 所需的一切都安装在 Docker 镜像中，所以只要安装 Docker 与 Git 即可。Docker 与 Git 的安装方法请参考第 2 章和 8.1.1 节。

首先安装 make。若已安装，则请跳过该部分。

### > Ubuntu

```
$ sudo apt-get install make
```

### > CentOS

```
$ sudo yum install make
```

从 GitHub 下载 Docker 源文件。也可以在下载完成后，根据需要转换分支或标签。

```
~$ git clone https://git@github.com/docker/docker
```

转到 docker 目录，然后运行 `sudo make build` 命令，如下所示。

```
~$ cd docker
~/docker$ sudo make build
```

稍后创建用于编译 Docker 的 `docker:master` 镜像（未转换 Git 分支且 `master` 未改变时）。接下来，运行 `sudo make binary` 命令。

```
~/docker$ sudo make binary
```

编译结束后，在 Docker 源目录下创建 bundles 目录，该目录中有编译好的 Docker 可执行文件。文件名格式为 docker-< 版本 >。

```
~/docker$ cd bundles/1.2.0-dev/binary
~/docker/bundles/1.2.0-dev/binary$ ls -al
total 13408
drwxr-xr-x 2 root root 4096 Sep 18 03:23 .
drwxr-xr-x 3 root root 4096 Sep 18 03:22 ..
lrwxrwxrwx 1 root root 16 Sep 18 03:23 docker -> docker-1.2.0-dev
-rwxr-xr-x 1 root root 13710473 Sep 18 03:23 docker-1.2.0-dev
-rw-r--r-- 1 root root 51 Sep 18 03:23 docker-1.2.0-dev.md5
-rw-r--r-- 1 root root 83 Sep 18 03:23 docker-1.2.0-dev.sha256
```

若使用包安装了 Docker，可以运行如下命令，用新的可执行文件替代系统中已经安装的 Docker 可执行文件。

```
~/docker/bundles/1.2.0-dev/binary$ sudo service docker stop
~/docker/bundles/1.2.0-dev/binary$ cp docker-1.2.0-dev $(type -P docker)
~/docker/bundles/1.2.0-dev/binary$ sudo service docker start
```

## 关注图灵教育 关注图灵社区 iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈 ……



QQ联系我们

读者QQ群: 218139230 164939616



微博联系我们

官方账号: @图灵教育 @图灵社区 @图灵新知

市场合作: @图灵袁野

写作本版书: @图灵小花 @图灵张霞

翻译英文书: @朱巍ituring @楼伟珊

翻译日文书或文章: @图灵日语编辑部

翻译韩文书: @图灵陈曦

电子书合作: @hi\_jeanne

图灵访谈/《码农》杂志: @刘敏ituring

加入我们: @王子是好人



微信联系我们



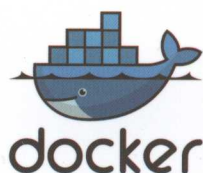
图灵教育  
turingbooks



图灵访谈  
ituring\_interview



# 基于Linux的轻量级容器， 在所有云服务实现快速部署！



## 应用部署系统，实现“一次构建，处处运行”

只要拥有Docker就能搭建安全的运行时环境，以便随时随地运行应用。借助Docker可以快速部署应用，不必面对系统不同导致的重复安装、设置与依赖性问题。虚拟机在系统兼容、迁移等方面存在许多制约因素，使用Docker则可不受任何束缚。用户可以在Amazon Web服务、谷歌云平台等平台自由迁移并发布应用。

## 无虚拟机负荷的轻量级虚拟环境

Hypervisor是对CPU、RAM、存储器等硬件的完全抽象，而Docker只对操作系统内核进行抽象。Docker共享操作系统并提供应用程序运行所需的虚拟化与隔离功能，所以既轻量又快捷。使用Docker可以对镜像创建、快照生成、初始化等进行快速处理。

## 面向开发人员与管理员的部署系统

只要构建运行时环境并打包，之后即可在所有机器反复运行。与虚拟机一样，应用在与主机隔离的环境中运行。开发人员可以将更多时间投入代码编写，管理员则可以将主要精力用于管理部署镜像，而不必逐个管理单个服务器，这样能够更好地保持系统一致性。采用Docker部署应用能够大大减少或杜绝开发、测试以及服务运营中系统的不一致性或兼容性问题。



ISBN 978-7-115-41962-0



9 787115 419620 >

ISBN 978-7-115-41962-0

定价：69.00元

图灵社区：iTuring.cn

热线：(010) 51095186 转 600

分类建议 计算机/Docker容器云

人民邮电出版社网址：www.ptpress.com.cn